Syllabus

1. Course Title:Computer Architecture

2. Course ID:

3. Course Units:

Lecture:30 hours

Seminar/ Exercise:15 hours

Total unit: 3 Credits

4. Expected Participants

Three-year students in Undergraduate Programs with having good English skills.

5. Prerequisites

Circuits and Electronics, Digital Electronics, Digital System, Analysis and Design of Digital Integrated Circuits, Computation Structures, Programming.

6. Course Objectives

The course is designed for undergraduate student. The course is intended to provide student the fundamentals of computer architecture and organization, the factors influencing the design of hardware and software elements to computer systems.

The goals of the course are to provide students with the basis knowledge of the followings:

- Introduction to Organization and Architecture Computer
- Overview of Computer Organization;
- Computer Arithmetic;
- Instruction-set Architecture;
- CPU Structure and Function;
- Control Unit Operation;
- Micro-Programming;
- Instruction Pipelining;
- Multilevel Memories;
- Cache Memory;
- Internal Memory;
- External Memory;
- Input/Output;
- Operating System Support;
- Virtual Memory;
- Advanced Architectures.

7. Course Description

This course is about the aspects of both Computer Architecture (attributes of a multilevel machine, a system visible to Micro-architecture, Micro-Programming level) and Computer Organization (the operational components and their interconnections in the system). Moreover, some advanced architectures will be presented.

This courser consists 16 lecture notes of 16 modules/sessions. Each module is designed to provide theoretical fundamental and practical exercises.

8. Student Duties

- Class attendance: ≥ 80%
- Homework

9. Assessment

- Mid-term grade: 0.4 (40%)

- Home exercise grading

- Mini-seminar or Mid-tem test

- Final exam : 0.6 (60%)

10. Course Materials:

Textbooks

[1]. William Stalling, Computer Organization and Architecture: Designing for Performance, 6th Edition, Prentice Hall, 2003.

This is the main textbook used in this course. Supplemental readings from selected papers may also be assigned; in which case, there may be a nominal photocopying charge, 2003

[2].Patterson-Hennessy, Computer Organization and Design (3th Edition)
[3]. Andrew S. Tannenbauum, Structured Computer Organization, 5th Edition, Prentice Hall, 2006.

Calendar

Course Calendar

Week	Topic	Reading	Work
1	Introduction to Organization and Architecture Computer Overview of Computer Organization	Course: Module 1 and Module 2Book [1]: Part 1Book [2]: Chapter 1	Course
2	Computer Arithmetic	Course: Module 3.Book [1]: Part III, chapter 9Book [2]: Chapter 5	Course and Exercise
3	Instruction Set: Characteristics and Functions	Course: Module 4.Book [1]: Part III, chapter 11	Course and Exercise
4	CPU Structure and Functions	Course: Module 5.Book [1]: Part III, chapter	Course and Exercise

		12Book [2]: Chapter 2	
5	Control Unit Operation	Course: Module 6. Book [1]: Part IV, chapter 16Book [2]: Chapter 4	Course
6	Microprogramming	Course: Module 7. Book [1]: Part IV, chapter 17Book [2]: Chapter 4	Course
7	Instruction Pipelining	Course: Module 8.Book [1]: Part III chapter 13 Book [2]: Chapter 4	Course and Exercise
8	Multilevel Memories	Course: Module 9.Book [2]: Chapter 2	Course and Exercise
9	Cache Memory	Course: Module 10.Book [1]: Part II, chapter 4Book [2]: Chapter 4	Course and Exercise
10	Internal Memory	Course:	Course

		Module 11.Book [1]: Part II, chapter 5Book [2]: Chapter 2	and Exercise
11	External Memory	Course: Module 12Book [1]: Part II, chapter 6Book [2]: Chapter 2	Course
12	Input/ Output	Course: Module 13Book [1]: Part II, chapter 7Book [2]: Chapter 2	Course and Exercise
13	Operating System Support	Course: Module 14Book [1]: Part II, chapter 8Book [2]: Chapter 6	Course and Exercise
14	Virtual Memory	Course: Module 15. Book [2]: Chapter 6	Course and Exercise
15	Advanced Architectures	Course: Module 16.Book [1]: Part V, chapter	Course and Exercise

Exercise

Exercise for module 3

Exercise 3.1

Find the following differences using twos complement arithmetic:

a) 111000

- 110011

b) 11001100

- 101110

c) 111100001111

-110011110011

d) 11000011

-11101000

Exercise 3.2

Given x=0101 and y =1010 in twos complement notation (i.e. x=4, y=6), compute the product p= x *y .

Exercise 3.3

Divide -145 by 13 in binary twos complement notation, using 12 bit words.

Exercise 3.4

Express the following number in IEEE 32-bit floating-point format:

a) -5 b) -6 c) -1.5 d) 384 e) 1/16 f) -1/32

Exercise 3.5

Consider a floating-point format with 8 bits for the biased exponent and 23 bits for the significant. Show the bit pattern for the following numbers in this format:

1. -720
2. 0.645

Exercise for module 4

Exercise 4.1

Let the address stored in the program counter be designated by the symbol X1. The instruction stored in X1 has an address part (operand reference) X2. The operand needed to execute the instruction is stored in the memory word with address X3. An index register contains the value X4. What is the relationship between these various quantities if the addressing mode of the instruction is

a) directb) indirectc) PC relative d) indexed

Exercise 4.2

An address field in an instruction contains decimal value 14. Where is the corresponding operand located for:

a) immediate addressing?

b) direct addressing?

c) indirect addressing?

d) register addressing?

e) register indirect addressing?

Exercise for module 5, 6

Exercise 5.1

If the last operation performed on a computer with an 8-bit word was an addition in which the two operands were 2 and 3, what would be the value of the following flags?

Carry

Zero

Overflow

Sign

Even parity

Exercise 5.2

There is only a two-stage pipeline (fetch, execute). Draw the timing diagram to show how many time units are needed for 4 instructions.

Exercise 6.1

Your ALU can add its two input registers, and it can logically complement the bits of either input register. But it cannot subtract. Numbers are to be stored in twos complement representation. List the micro-operations your control unit must perform to cause a subtraction.

Exercises for module 8

Exercise 8.1: A computer with a five-stage pipeline deals with conditional braches by stalling for the next three cycles after hitting one. How much does stalling hurt the performance of 20% of all instructions are conditional branches? Ignore all sources of stalling except conditional branches.

Exercise 8.2: Suppose that a computer prefetches up to 20 instructions in advance. However, on the average, four of these are conditional branches, each with a probability of 90 % of being predicted correctly. What is the probability that the prefetching is on the right track?

Exercise 8.3: Normally, dependences cause trouble with pipelined CPUs. Are there any optimizations that can e done with WAW dependences that might actually improve matters? What?

Exercise for module 9

Exercise 9.1: A computer with a 32-bit wide data bus uses 1M x 1 dynamic RAM memory chips. What is the smallest memory (in bytes) that computer can have?

Exercise 9.2: A 32-bit CPU with address line A2-A31 requires all memory references to be aligned. That is, words have to be addressed al multiples of 4 bytes, and haft-words have to be addressed at even bytes. Bytes can be anywhere. How many legal combinations are there for memory reads, and how many pins are needed to express them? Give two answers and make a case for each one.

Exercises for module 10

Exercise 10.1: A computer has 1 two-level cache. Suppose that 60% of the memory references hit on the first level cache, 35% hit on second level and 5% miss. The access time are 5ns, 15ns and 60ns, respectively, where the time for the level 2 cache and memory start counting at the moment it is known that they are needed (e.g. a level 2 cache access does not even start until the level 1 cache miss occurs). What is the average access time?

Exercise 10.2: Write a simulator for a 1-way direct mapped cache. Make the number of entries and line size parameters of the simulation. Experiment with it and report on your findings.

Exercises for module 13

Exercise 13.1: Calculate the bus bandwidth needed to display a VGA (640 x 480) true color movie at 30 frames/sec. Assume that the data must pass over the bus twice, one from the CDROM to the memory and once from the memory to the screen.

Exercise 13.2: A computer has instruction that each require two bus cycles, one to fetch the instruction and one to fetch the data. Each bus cycle takes 10 ns and each instruction takes 20 ns (i.e. the internal processing time is negligible). The computer also has a disk with 2048 512-bytes sectors per track. Disk rotation time is 5 ms. To what percent of its normal speed is the computer reduced during a DMA transfer if each 32-bit DMA transfer takes one bus cycle?

Exercises for module 14

Exercise 14.1 Operating systems that allow memory-mapped files always require files to be mapped at page boundaries. For example, with 4-KB page, a file can be mapped in starting at virtual address 4096, but not starting at virtual address 5000. Why?

Exercise 14.2: In some way, caching and paging are very similar. In both case there are two levels of memory (the cache and main memory in the former, and main memory and disk in the latter). . We look at some of the arguments in favor of larger disk pages and small disk pages. Do the same arguments hold for cache line size?

Exercise for module 15

Exercise 15.1: It is common to test the page replacement algorithms by simulation. For this exercise, you are to write a simulator for the page-based virtual memory for a machine with 64 1-KB pages. The simulator should maintain a single table of 64 entries, one per page, containing the physical page number corresponding to that virtual page. Generate a file consisting of random address and test performance for both LRU and FIFO algorithms.

Exercises for module 16

Exercise 16.1: Consider a multiprocessor using a shared bus. What happens if two processors try to access the global memory at exactly the same time?

Module 1: Introduction to Organization and Architecture Of Computer This module presents an introduction and a background to helps understanding the fundamental concepts of Organization and Architecture Computer. The module includes(1)Fundamental concepts of Organization and Architecture, (2)Structure and Function of computer, and (3)Brief History of Computers. Readings and Resources: Part I, Computer Organization and Architecture: Designing for Performance, 6th Edition by William Stalling, textbook.

Organization and Architecture

In describing computer system, a distinction is often made between **computer architecture** and **computer organization**. Although it is difficult to give precise definition for these terms, a consensus exists about the general areas covered by each.

Computer architecture refers to those attributes of a system visible to a programmer, or put another way, those attributes that have a direct impact on the logical execution of a program.

Computer organization refers to the operational units and their interconnection that realize the architecture specification.

Examples of architecture attributes include the instruction set, the number of bit to represent various data types (e.g.., numbers, and characters), I/O mechanisms, and technique for addressing memory. Organization attributes include those hardware details transparent to the programmer, such as control signals, interfaces between the computer and peripherals, and the memory technology used.

As an example, it is an architectural design issue whether a computer will have a multiply instruction. It is an organizational issue whether that instruction will be implemented by a special multiply unit or by a mechanism that makes repeated use of the add unit of the system. The organization decision may be bases on the anticipated frequency of use of the multiply instruction, the relative speed of the two approaches, and the cost and physical size of a special multiply unit.

Historically, and still today, the distinction between architecture and organization has been an important one. Many computer manufacturers offer a family of computer model, all with the same architecture but with differences in organization. Consequently, the different models in the family have different price and performance characteristics. Furthermore, an architecture may survive many years, but its organization changes with changing technology.

Structure and Function

A computer is a complex system; contemporary computers contain million of elementary electronic components. How, then, can one clearly describe them? The key is to recognize the hierarchic nature of most complex system. A hierarchic system is a set of interrelated subsystem, each of the later, in turn, hierarchic in structure until we reach some lowest level of elementary subsystem.

The hierarchic nature of complex systems is essential to both their design and their description. The designer need only deal with a particular level of the system at a time. At each level, the system consists of a set of components and their interrelationships. The behavior at each level depends only on a simplified, abstracted characterization of the system at the nest lower level. At each level, the designer is concerned with structure and function:

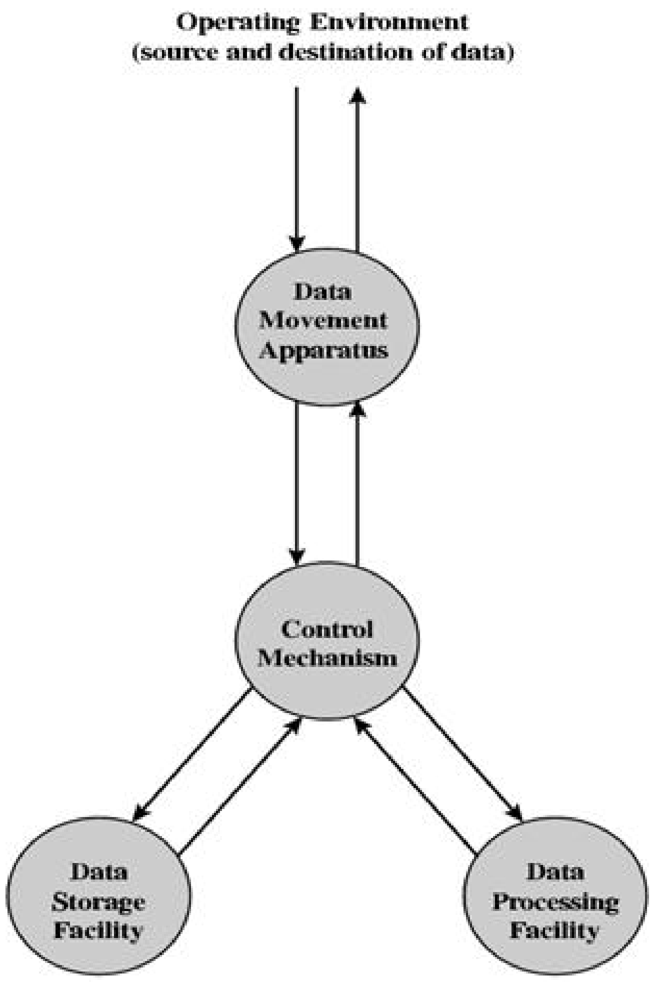
- Structure: The way in which the components are interrelated.
- Function: The operation of each individual component as part of the structure.

In term of description, we have to choices: starting ant the bottom and building up to a complete description or with a top view and decomposing the system, describing their structure and function, and proceed to successively lower layer of the hierarchy. The approach taken is this course follows the latter.

Functions

In general terms, there are four main functions of a computer:

- Data processing
- Data storage
- Data movement
- Control



A functional view of the computer

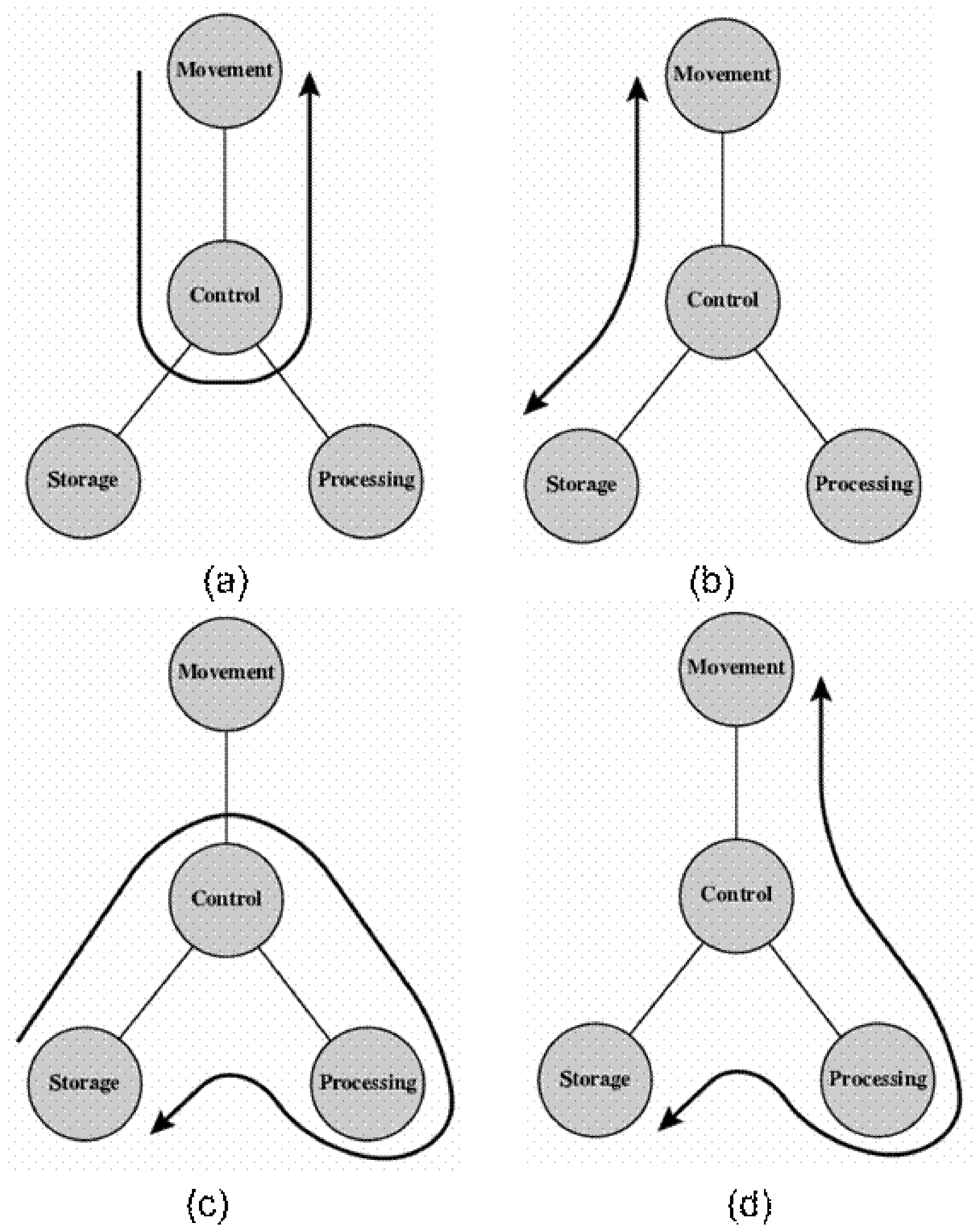
The computer, of course, must be able to process data. The data may take a wide variety of forms, and the range of processing requirements is broad. However, we shall see that there are only a few fundamental methods or types of data processing.

It is also essential that a computer store data. Event if the computer is processing data on the fly (i.e., data come in and get processed, and the results go right out), the computer must temporarily store at least those pieces of data that are being worked on at any given moment. Thus, there is at least a short-term data storage function. Files of data are stored on the computer for subsequent retrieval and update.

The computer must be able to move data between itself and the outside world. The computer's operating environment consists of devices that serve as either sources or destinations of data. When data are received from or delivered to a device that is directly connected to the computer, the process id known as input-output (I/O), and the device is referred to as a peripheral. When data are moved over longer distances, to or from a remote device, the process is known as data communications.

Finally, there must be control of there three functions. Ultimately, this control is exercised by the individual who provides the computer with instructions. Within the computer system, a control unit manages the computer's resources and orchestrates the performance of its functional parts in response to those instructions.

At this general level of discussion, the number of possible operations that can be performed is few. The [link] depicts the four possible types of operations. The computer can function as a data movement device ([link] (a)), simply transferring data from one peripheral or communications line to another. It can also function as a data storage device ([link](b)), with data transferred from the external environment to computer storage (read) and vice versa (write). The final two diagrams show operations involving data processing, on data either in storage or in route between storage and the external environment.

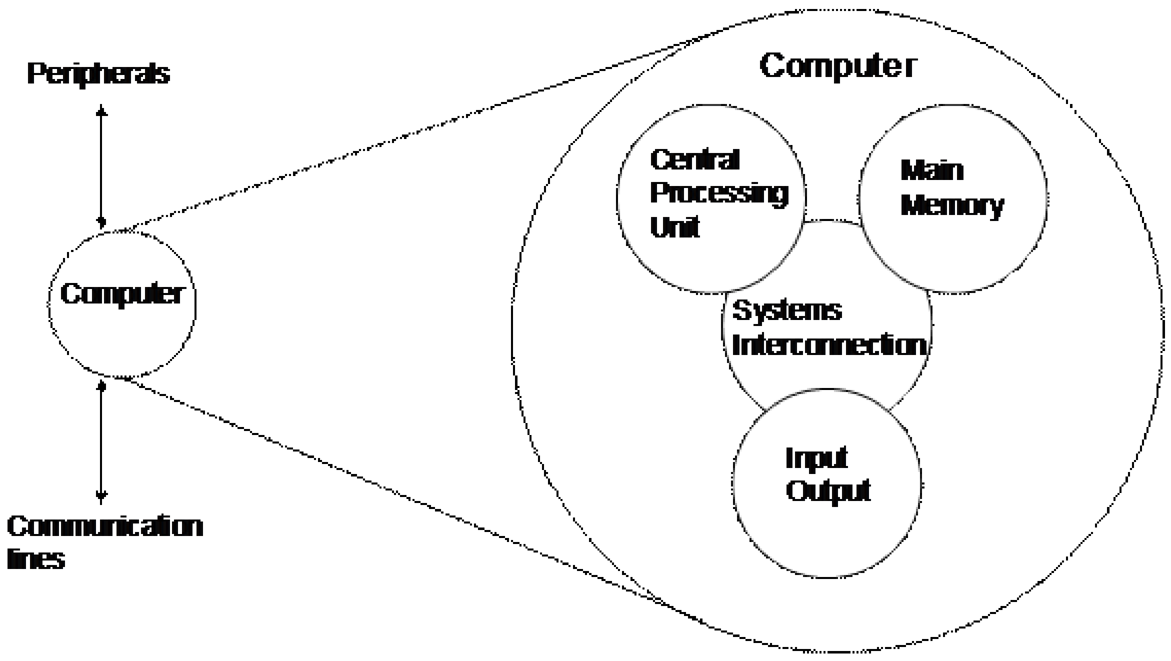


Possible computer operations

Structure

[link] is the simplest possible depiction of a computer. The computer is an entity that interacts in some fashion with its external environment. In general, all of its linkages to the external environment can be classified as peripheral devices or communication lines. We will have something to say about both types of linkages.

- Central Processing Unit (CPU): Controls the operation of the computer and performs its data processing functions. Often simply referred to as processor.
- Main Memory: Stores data.
- I/O: Moves data between the computer and its external environment.
- System Interconnection: Some mechanism that provides for communication among CPU, main memory, and I/O.

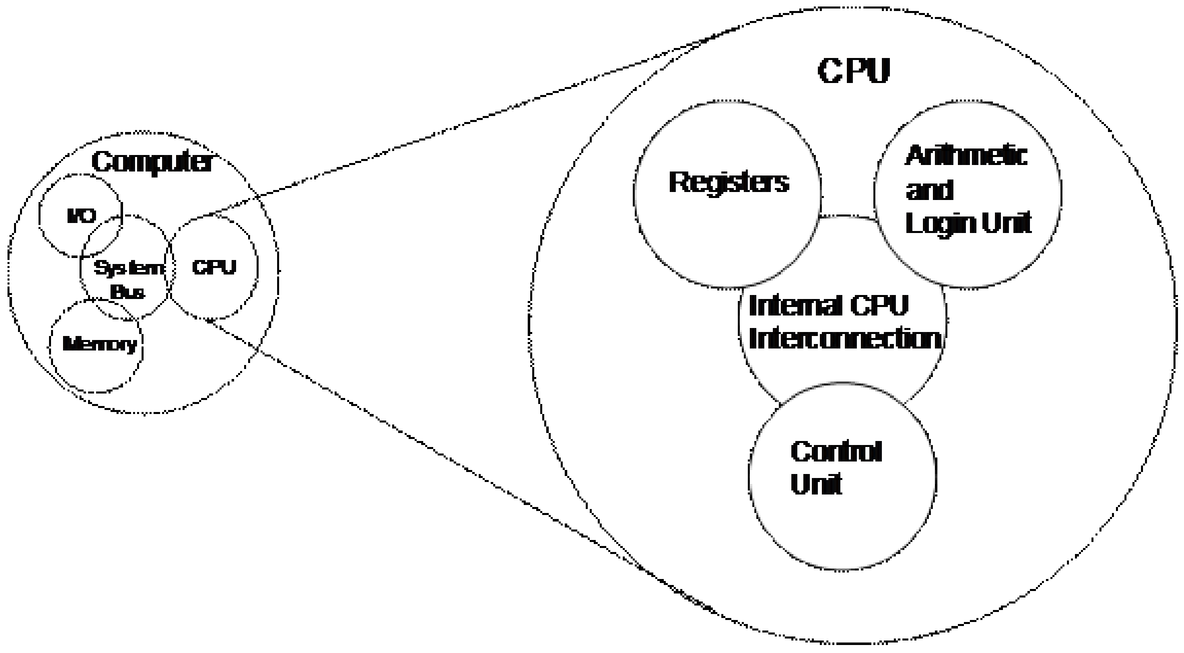


The computer: top-level structure

There may be one or more of each of the above components. Traditionally, there has been just a single CPU. In recent years, there has been increasing use of multiple processors, in a single system. Each of these components will be examined in some detail in later lectures. However, for our purpose, the most interesting and in some ways the most complex component is the CPU; its structure is depicted in [link]. Its major structural components are:

- Control Unit (CU): Controls the operation of the CPU and hence the computer.
- Arithmetic and Logic Unit (ALU): Performs computer's data processing functions.
- Register: Provides storage internal to the CPU.
- CPU Interconnection: Some mechanism that provides for communication among the control unit, ALU, and register.

Each of these components will be examined in some detail in next lectures.



The CPU

Brief history of Computers

The first Generation: Vacuum Tubes

- ENIAC

The ENIAC (Electronic Numerical Integrator And Computer), designed by and constructed under the supervision of Jonh Mauchly and John Presper Eckert at the University of Pennsylvania, was the world’s first general-purpose electronic digital computer. The project was a response to U.S. wartime needs. Mauchly, a professor of electrical engineering at the University of Pennsylvania and Eckert, one of his graduate students, proposed to build a general-purpose computer using vacuum tubes. In 1943, this proposal was accepted by the Army, and work began on the ENIAC. The resulting machine was enormous, weighting 30 tons, occupying 15,000 square feet of floor space, and containing more than 18,000 vacuum tubes. When operating, it consumed 140 kilowatts of power. It was aloes substantially faster than any electronic-mechanical computer, being capable of 5000 additions per second.

The ENIAC was decimal rather than a binary machine. That is, numbers were represented in decimal form and arithmetic was performed in the decimal system. Its memory consisted of 20 “accumulators”, each capable of holding a 10-digit decimal number. Each digit was represented by a ring of 10 vacuum tubes. At any time, only one vacuum tube was in the ON state, representing one of the 10 digits. The major drawback of the ENIAC was that it had to be programmed manually by setting switches and plugging and unplugging cables.

The ENIAC was completed in 1946, too late to be used in the war effort. Instead, its first task was to perform a series of complex calculations that were used to help determine the feasibility of the H-bomb. The ENIAC continued to be used until 1955.

- The von Neumann Machine

As was mentioned, the task of entering and altering programs for the ENIAC was extremely tedious. The programming process could be facilitated if the program could be represented in a form suitable for storing in memory alongside the data. Then, a computer could get its instructions by reading them from memory, and a program could be set of altered by setting the values of a portion of memory.

This idea, known as the **Stored-program concept**, is usually attributed to the ENIAC designers, most notably the mathematician John von Neumann, who was a consultant on the ENIAC project. The idea was also developed at about the same time by Turing. The first publication of the idea was in a 1945 proposal by von Neumann for a new computer, the EDVAC (Electronic Discrete Variable Computer).

In 1946, von Neumann and his colleagues began the design of a new stored-program computer, referred to as the IAS computer, at the Princeton Institute for Advanced Studies. The IAS computer, although not completed until 1952, is the prototype of all subsequent general-purpose computers. [link] shows the general structure of the IAS computer. It consists of:

- A main memory, which stores both data and instructions.
- An arithmetic-logical unit (ALU) capable of operating on binary data.
- A control unit, which interprets the instructions in memory and causes them to be executed.
- Input and output (I/O) equipment operated by the control unit.

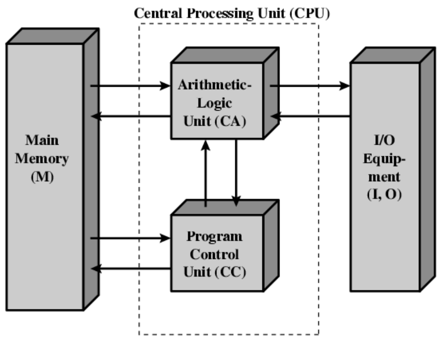

Central Processing Unit (CPU)
Main Memory (M)
Arithmetic-Logic Unit (CA)
Program Control Unit (CC)
I/O Equip-ment (I, O)

Figure 2.1 Structure of the IAS Computer

Structure of the IAS computer

- Commercial Computers

The 1950s saw the birth of the computer industry with two companies, Sperry and IBM, dominating the marketplace.

In 1947, Eckert and Mauchly formed the Eckert-Maunchly computer Corporation to manufacture computers commercially. Their first successful machine was the UNIVAC I (Universal Automatic Computer), which was commissioned by the Bureau of the Census for the 1950 calculations. The Eckert-Maunchly Computer Corporation became part of the UNIVAC division of Sperry-Rand Corporation, which went on to build a series of successor machines.

The UNIVAC II, which had greater memory capacity and higher performance than the UNIVAC I, was delivered in the late 1950s and illustrates several trends that have remained characteristic of the computer industry. First, advances in technology allow companies to continue to build larger, more powerful computers. Second, each company tries to make its new machines upward compatible with the older machines. This means that the programs written for the older machines can be executed on the new machine. This strategy is adopted in the hopes of retaining the customer base; that is, when a customer decides to buy a newer machine, he is likely to get it from the same company to avoid losing the investment in programs.

The UNIVAC division also began development of the 1100 series of computers, which was to be its bread and butler. This series illustrates a distinction that existed at one time. The first model, the UNIVAC 1103, and its successors for many years were primarily intended for scientific applications, involving long and complex calculations. Other companies concentrated on business applications, which involved processing large amounts of text data. This split has largely disappeared but it was evident for a number of years.

IBM. which was then the major manufacturer of punched-card processing equipment, delivered its first electronic stored-program computer, the 701, in 1953. The 70l was intended primarily for scientific applications. In 1955, IBM introduced the companion 702 product, which had a number of hardware features that suited it to business applications. These were the first of a long series of 700/7000 computers that established IBM as the overwhelmingly dominant computer manufacturer.

The Second generation: Transistors

The first major change in the electronic computer came with the replacement of the vacuum tube by the transistor. The transistor is smaller, cheaper, and dissipates less heal than a vacuum tube but can be used in the same way as a vacuum tube to construct computers. Unlike the vacuum tube, which requires wires, metal plates, a glass capsule, and a vacuum, the transistor is a solid-state device, made from silicon.

The transistor was invented at Bell Labs in 1947 and by the 1950s had launched an electronic revolution. It was not until the late 1950s, however, that fully transistorized computers were commercially available. IBM again was not the first company to deliver the new technology. NCR and. more successfully. RCA were the front-runners with some small transistor machines. IBM followed shortly with the 7000 series.

The use of the transistor defines the second generation of computers. It has become widely accepted to classify computers into generations based on the fundamental hardware technology employed (Table 2.2). Each new generation is characterized by greater processing performance, larger memory capacity, smaller size than the previous one.

The 3th generation

A single, self-contained transistor is called a discrete component. Throughout the 1950s and early 1960s, electronic equipment was composed largely of discrete com¬ponents—transistors, resistors, capacitors, and so on. Discrete components were manufactured separately, packaged in their own containers, and soldered or wired together onto circuit boards, which were then installed in computers, oscilloscopes, and other electronic equipment. Whenever an electronic device called for a transistor, a little lube of metal containing a pinhead-sized piece of silicon had to be soldered to a circuit hoard. The entire manufacturing process, from transistor to circuit board, was expensive and cumbersome. These facts of life were beginning to create problems in the computer industry. Early second-generation computers contained about 10,000 transistors. This figure grew to the hundreds of thousands, making the manufacture of newer, more power¬ful machines increasingly difficult. In 1958 came the achievement that revolutionized electronics and started the era of microelectronics: the invention of the integrated circuit. It is the integrated circuit that defines the third generation of computers. Perhaps the two most important members of the third generation are the IBM System/360 and the DEC PDP-8.

The 4th generation

Module 2: Top-Level View of Computer Organization

1. Computer Component

Contemporary computer designs are based on concepts developed by John von Neumann at the Institute for Advanced Studies Princeton. Such a design is referred to as the von Neumann architecture and is based on three key concepts:

- Data and instructions are stored in a single read -write memory.
- The contents of this memory are addressable by location, without regard to the type of data contained there.
- Execution occurs in a sequential fashion (unless explicitly modified) from one instruction to the next.

Suppose that there is a small set of basic logic components that can be com-bined in various ways to store binary data and to perform arithmetic and logical operations on that data. If there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed. We can think of the process of connecting the various components in the desired configuration as a form of programming. The resulting "program"' is in the form of hardware and is termed a hardwired program.

Now consider other alternative. Suppose we construct a general-purpose configuration of arithmetic and logic functions, this set of hardware will perform various functions on data depending on control signals applied to the hardware. In the original case of customized hardware, the system accepts data and produces results (Figure 1a). With general-purpose hardware, the system accepts data and control signals and produces results. Thus, instead of rewiring the hardware for each new program, The programmer merely needs to supply a new set of control signals.

How shall control signals be supplied? The answer is simple but subtle. The entire program is actually a sequence of steps. At each step, some arithmetic or logical operation is performed on some data. For each step, a new set of control signals is needed. Let us provide a unique code for each possible set of control signals, and let us add to the general-purpose hardware a segment that can accept a code and generate control signals (Figure 1b).

Programming is now much easier. Instead of rewiring the hardware for each new program, all we need to do is provide a new sequence of codes. Each code is, in effect, an instruction, and part of the hardware interprets each instruction and generates control signals. To distinguish this new method of programming, a sequence of codes or instructions is called software.

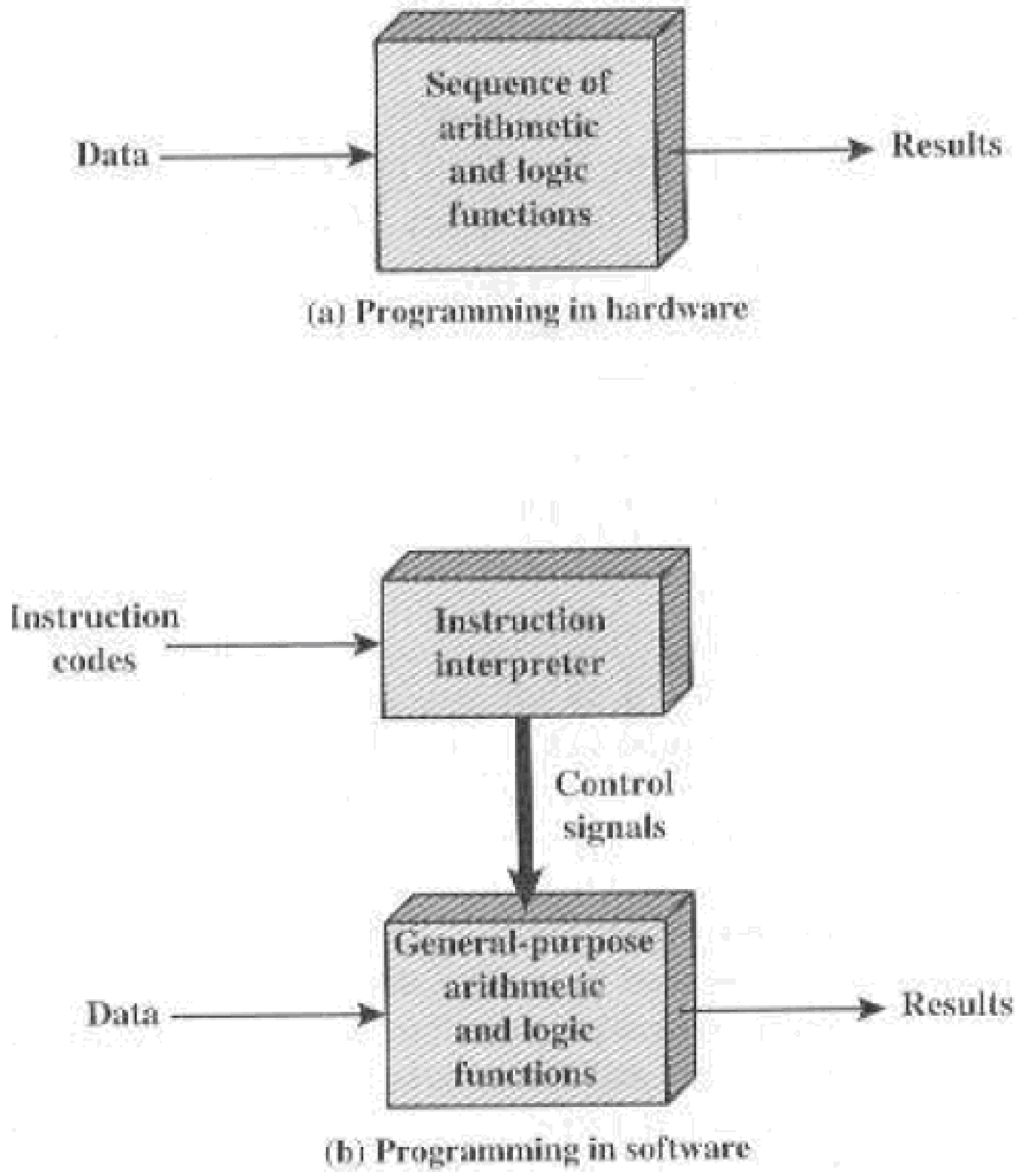

Data
Sequence of arithmetic and logic functions
Results
(a) Programming in hardware
Instruction codes
Instruction interpreter
Control signals
Data
General-purpose arithmetic and logic functions
Results
(b) Programming in software

Figure 1 Hardware and software approaches

Figure 1b indicates two major components of the system: an instruction interpreter and a module of general-purpose arithmetic and logic functions. These two constitute the CPU.

Several other components are needed to yield a functioning computer. Data and instructions must be put into the system. For this we need some sort of input module. This module contains basic components for accepting data and instructions in some form and converting them into an internal form of signals usable by the system. A means of reporting results is needed, and this is in the form of an output module. Taken together, these are referred to as I/O components.

One more component is needed. An input device will bring instructions and data in sequentially, but a program is not invariably executed sequentially: it may jump around. Similarly, operations on data may require access to more than just one element at a lime in a predetermined sequence. Thus, There must be a place to store temporarily both instructions and data. That module is called memory, or main memory to distinguish it from external storage or peripheral devices. Von Neumann pointed out that the same memory could be used to store both instructions and data.

Figure 2 illustrates these top-level components and suggests the interactions among them. The CPU exchanges data with memory. For this purpose, it typically makes use of two internal (to the CPU) registers: a memory address register (MAR), which specifies the address in memory for the next read or write, and a memory buffer register (MBR), which contains the data to be written into memory or receives the data read from memory. Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the CPU.

A memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a binary number that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to CPU and memory, and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.

Having looked briefly al these major components, we now turn to an overview of how these components function together to execute programs.

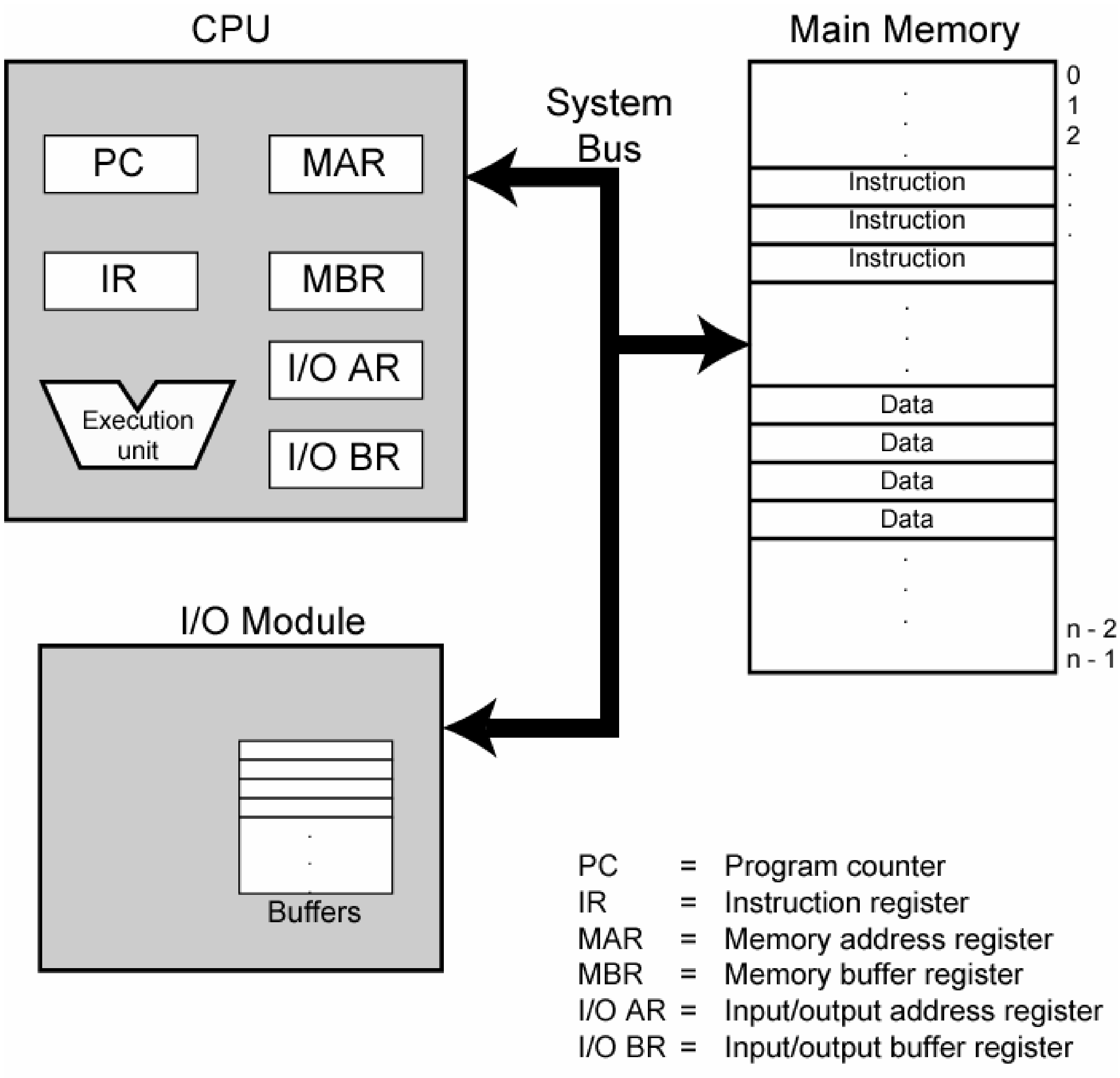


Figure 2 Computer components: Top-level view

2. Computer Function

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. In its simplest form, instruction processing consists of two steps: The processor reads (fetches) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. The Instruction execution may involve several operations and depends on the nature of the instruction.

The processing required for a single instruction is called an instruction cycle. Using the simplified two-step description given previously, the instruction cycle is depicted in Figure 3

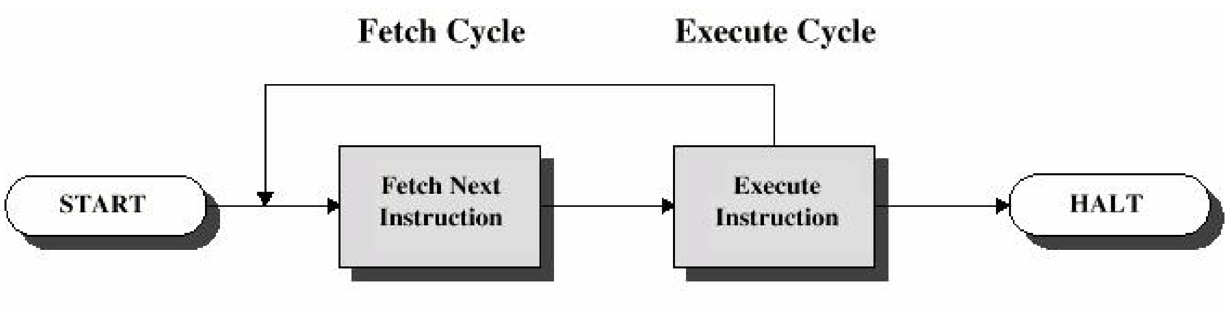


Figure 3: Basic instruction cycle

The two steps are referred to as the fetch cycle and the execute cycle. Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

2.1 Instruction Fetch and Execute

Fetch Cycle:

- Program Counter (PC) holds address of next instruction to fetch
- Processor fetches instruction from memory location pointed to by PC
- Increment PC (Unless told otherwise)
- Instruction loaded into Instruction Register (IR)
- Processor interprets instruction and performs required actions

At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next. Unless told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence. The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action.

Execute Cycle:

In general, the required actions fall into four categories:

- Processor-memory: Data may be transferred from processor to memory or from memory to processor.
- Processor-I/O: Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- Data processing: The processor may perform some arithmetic or logic operation on data.
- Control: An instruction may specify that the sequence of execution be altered.
- An instruction’s execution may involve a combination of these actions.

Instruction Cycle State Diagram:

Figure 4 provides a more detailed look at the basic instruction cycle. The figure is in the form of a state diagram. For any given instruction cycle, some stales may be null and others may be visited more than once. The states can be described as follows:

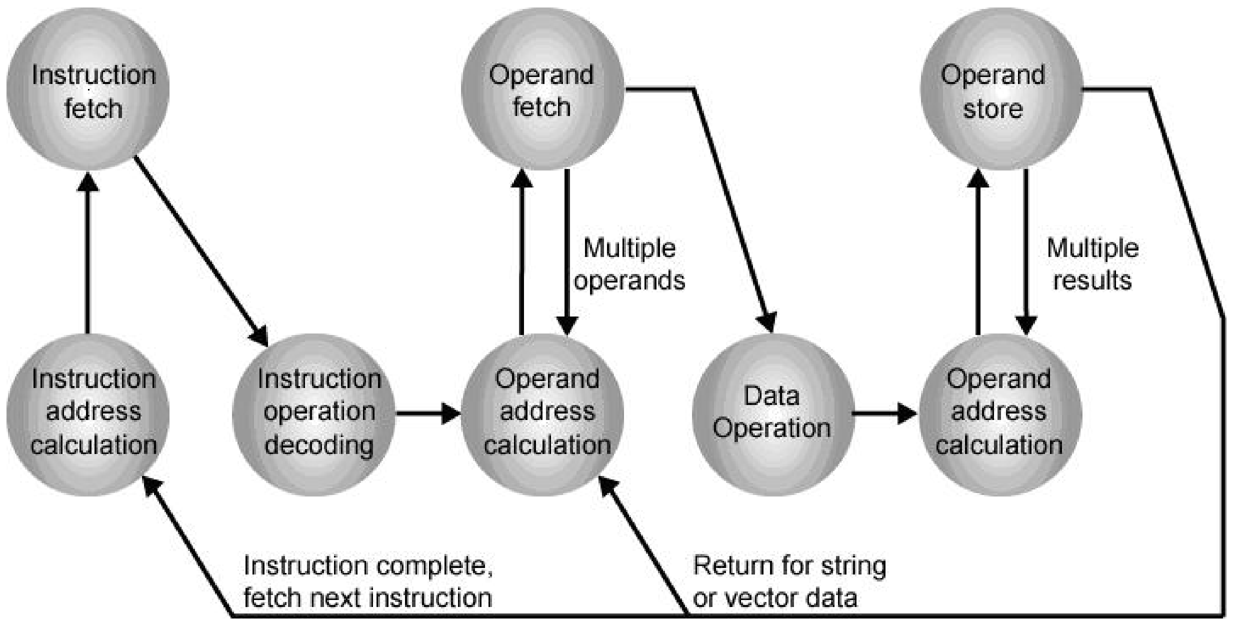

Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data

Figure 4: Instruction cycle state diagram

- Instruction address calculation (iac): Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.
- Instruction fetch (if): Read instruction from its memory location into the processor.
- Instruction operation decoding (iod): Analyze instruction to determine type of operation to he performed and operand(s) to be used.
- Operand address calculation (oac): If the operation involves reference to an operand in memory or available via I/O. then determine the address of the operand.
- Operand fetch (of): Fetch the operand from memory or read it in from I/O,
- Data operation (do): Perform the operation indicated in the instruction.
- Operand store (os): Write the result into memory or out to I/O

2.2 Interrupts

Virtually all computers provide a mechanism called Interrupt, by which other modules (I/O. memory) may interrupt the normal processing of the processor. Interrupts are provided primarily as a way to improve processing efficiency.

For example, most external devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of Figure 3. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many hundreds or even thousands of instruction cycles that do not involve memory. Clearly, this is a very wasteful use of the processor. The figure 5a illustrates this state of affairs.

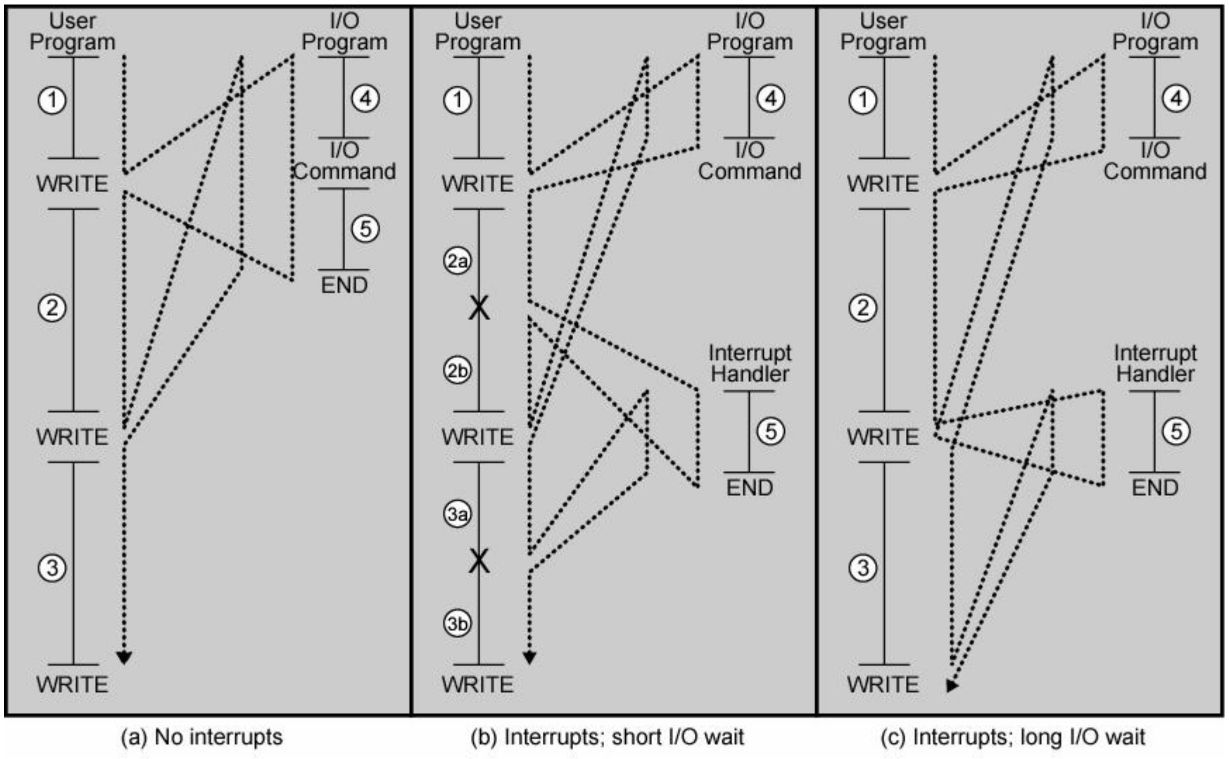


Figure 5: Program flow control without and with interrupts

The user program (depicted in figure 5a) performs a series of WRITE calls interleaved with processing. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls arc to an I/O program that is a System utility and that will perform the actual I/O operation. The I/O program consists of three sections:

- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
- The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically poll the device). The program might wail by simply repeatedly performing a test operation to determine if the I/O operation is done.
- A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

Because the I/O operation may lake a relatively long time to complete. The I/O program is hung up wailing for the operation to complete; hence. The user program is slopped at the point of the WRITE call for some considerable period of time

Interrupts and the Instruction Cycle

With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in Figure 5b. As before, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced, that is, when it is ready to accept more data from the processor, the I/O module for that external device sends an interrupt request signal to the processor. The processor responds by suspending operation of the current program, branching off to a program to service that particular I/O device, known as an interrupt handler, and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by an asterisk in Figure 5b.

From the point of view of the user program, an interrupt is just that: an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure 6). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the operating system are responsible for suspending the user program and then resuming it at the same point.

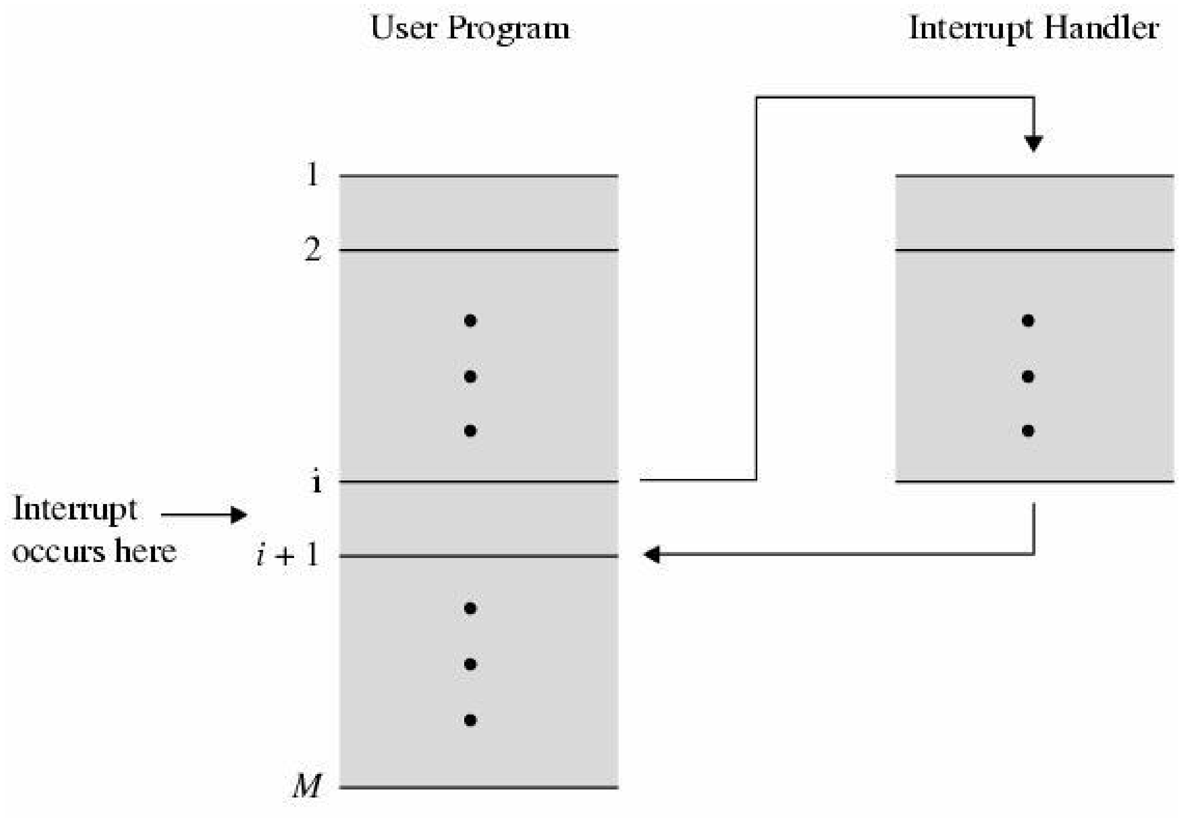


Figure 6: The transfer of control via interrupt

To accommodate interrupts, an interrupt cycle is added to the instruction cycle, as shown in Figure 7. In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts arc pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

- It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
- It sets the program counter to the starting address of an interrupt handler routine.

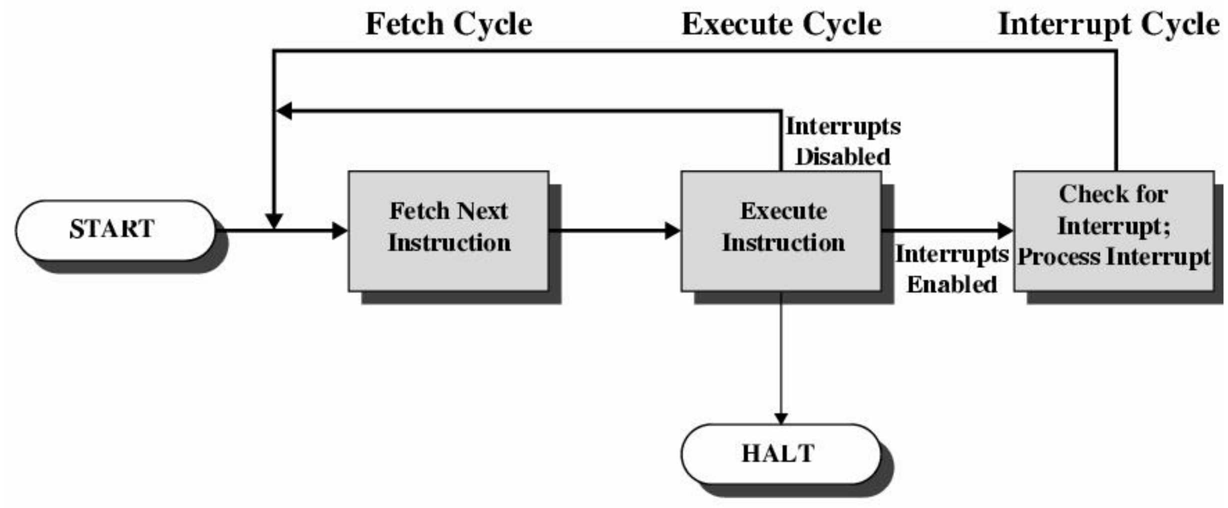


Figure 7: Instruction Cycle with Interrupts.

The processor now proceeds to the fetch cycle and fetches the first instruc-tion in the interrupt handler program, which will service the interrupt. The interrupt handler program is generally part of the operating system. Typically, this program determines the nature of the interrupt and performs whatever actions are needed. In the example we have been using, the handler determines which I/O module generated the interrupt, and may branch to a program that will write more data out to that I/O module. When the interrupt handler routine is completed, the processor can resume execution of the user program at the point of interruption. Figure 8 shows a revised instruction cycle state diagram that includes interrupt cycle processing.

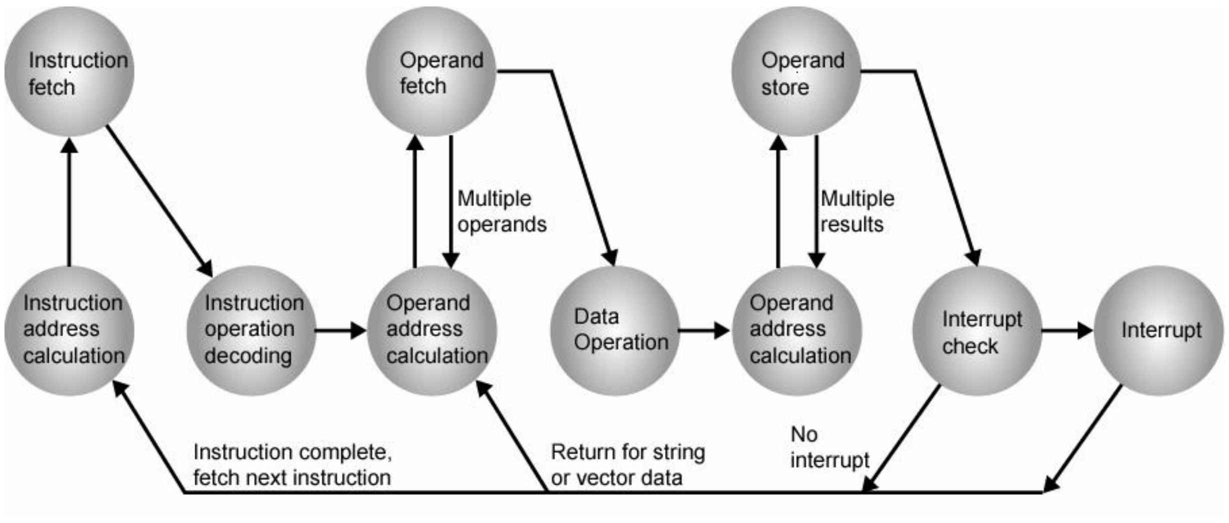

Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Interrupt check
Interrupt
No interrupt
Instruction complete, fetch next instruction
Return for string or vector data

Figure 8: Instruction cycle state diagram with interrupt

Multiple Interrupts

In some cases, multiple interrupts can occur. For example, a program may be receiving data from a communications line and printing results. The printer will generate an interrupt every lime that it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives. I he unit could either be a single character or a block, depending on the nature of the communications discipline. In any case, it is possible for a communications interrupt to occur while a printer interrupt is being processed. Two approaches can be taken to dealing with multiple interrupts:

- Disabling interrupts while an interrupt is being processed.
- Defining priorities for interrupts.

The first is to disable interrupts while an interrupt is being processed. A disabled interrupt simply means that the processor can and will ignore that interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be cheeked by the processor after the processor has enabled interrupts. Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately. After the interrupt handler routine completes, interrupts are enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred. This approach is nice and simple, as interrupts are handled in strict sequential order (Figure 2.10).

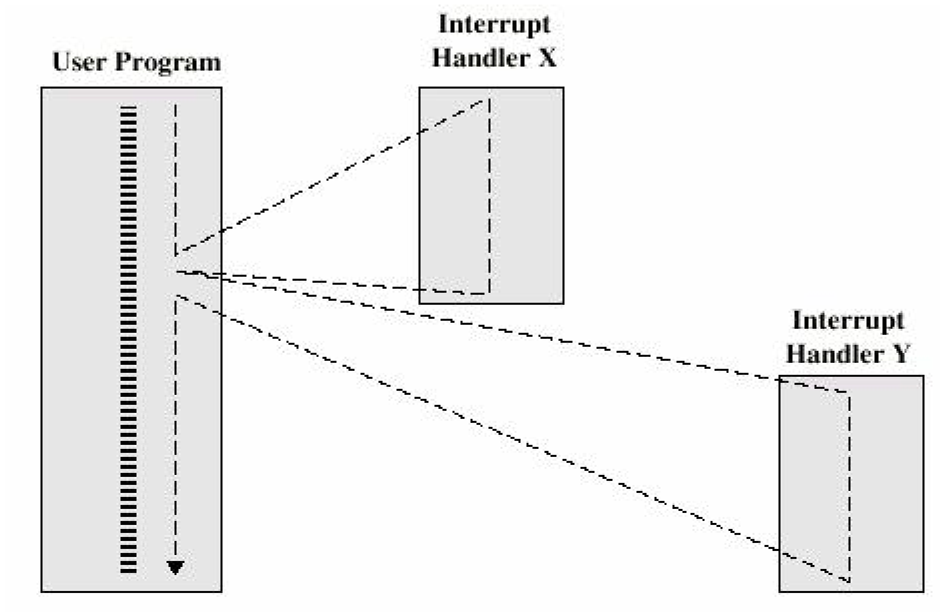


Figure 9: Sequential interrupt processing

The second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be itself interrupted (Figure 10)

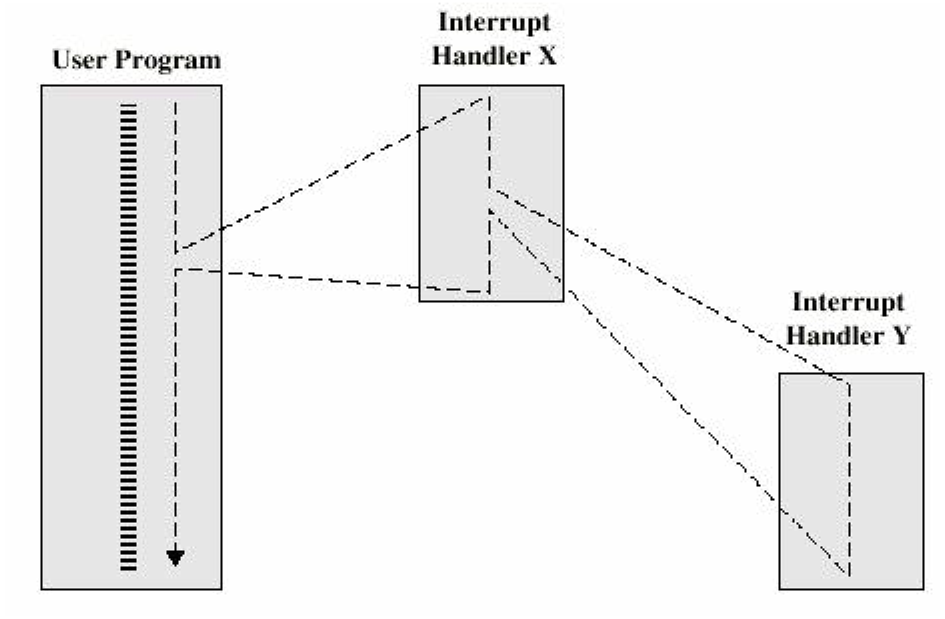


Figure 10: Transfer of Control with Multiple Interrupts

3. Interconnection Structures

A computer consists of a set of components or modules of three basic types (processor, memory, I/O) that communicate with each other. In effect, a computer is a network of basic modules. Thus, there must be paths for connecting the modules.

The collection of paths connecting the various modules is called the interconnection structure. The design of this structure will depend on the exchanges that must be made between modules.

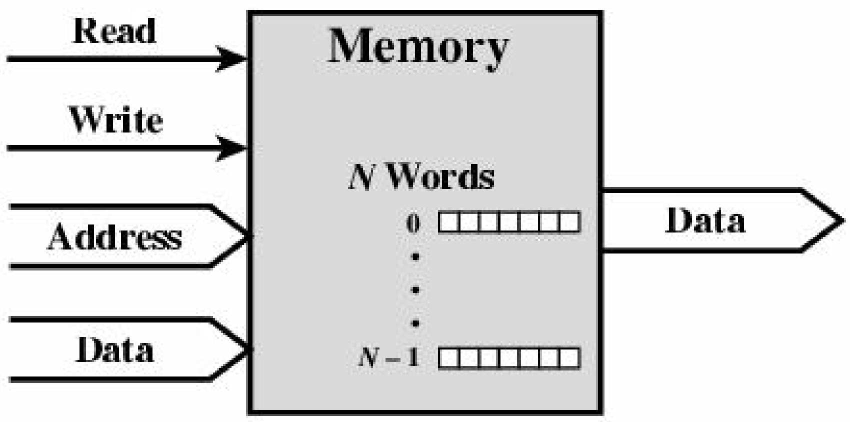
Figure 11 suggests the types of exchanges that are needed by indicating the major forms of input and output for each module type:

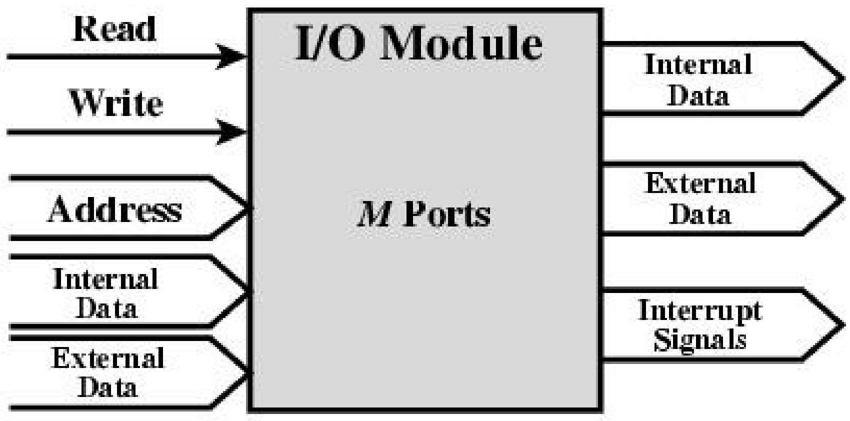
- Memory
- Input/Output
- CPU

The interconnection structure must support the following types of transfers:

- Memory to processor: The processor reads an instruction or a unit of data from memory.
- Processor to memory: The processor writes a unit of data to memory.
- I/O to processor: The processor reads data from an I/O device via an I/O module.
- Processor to I/O: The processor sends data to the I/O device.
- I/O to or from memory: For these two cases, an I/O module is allowed to exchange data directly with memory, without going through the processor, using direct memory access (DMA).

Over the years, a number of interconnection structures have been tried. By far the most common is the bus and various multiple-bus structures.


Read
Write
Address
Data
Memory
N Words
0
N – 1
Data


Read
Write
Address
Internal Data
External Data
I/O Module
M Ports
Internal Data
External Data
Interrupt Signals

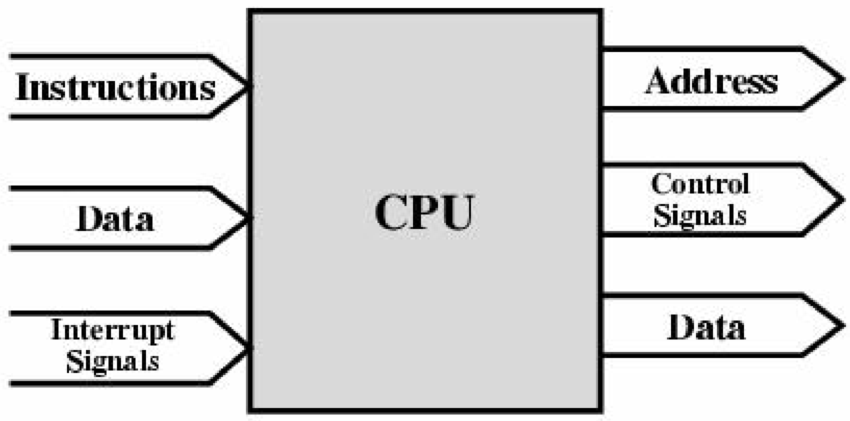

Instructions
Data
Interrupt Signals
CPU
Address
Control Signals
Data

Figure 11 Computer Modules

4. Bus Interconnection

A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium. Multiple devices connect to the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus (broadcast). If two devices transmit during the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.

Typically, a bus consists of multiple communication pathways, or lines. Each line is capable of transmitting signals representing binary 1 and binary 0. Overtime, a sequence of binary digits can be transmitted across a single line. Taken together, several lines of a bus can be used to transmit binary digits simultaneously (in parallel). For example, an 8-bil unit of data can be transmitted over eight bus lines.

Computer systems contain a number of different buses that provide pathways between components at various levels of the computer system hierarchy. A bus that connects major computer components (processor, memory, I/O) is called a system bus. The most common computer interconnection structures are based on the use of one or more system buses.

Bus Structure

A system bus consists, typically, of from about 50 to hundreds of separate lines. Each line is assigned a particular meaning or function. Although there are many different bus designs, on any bus the lines can be classified into three functional groups (Figure 12): data, address, and control lines. In addition, there may he power distribution lines that supply power to the attached modules.

The data lines (data bus):

- Provide a path for moving, data between system modules. These lines, collectively, are called the data bus.
- The width of the data bus: The data bus may consist of from 32 to hundreds of separate lines, the number of lines being referred to as the width of the data bus. Because each line can carry only 1 bit at a time, the number of lines determines how many bits can be transferred at a lime. The width of the data bus is a key factor in determining overall system performance. For example, if the data bus is 8 bits wide and each instruction is 16 bits long, then the processor must access the memory module twice during each instruction cycle.

The address lines (address bus):

- Address lines are used to designate the source or destination of the data on the data bus. For example, if the processor wishes to read a word (8, 16. or 32 bits) of data from memory, it puts the address of the desired word on the address lines.
- The width of the address bus: determines the maximum possible memory capacity of the system. Furthermore, the address lines are generally also used to address I/O ports.

The control lines (control bus):

- Control bus are used to control the access to and the use of the data and address lines. Because the data and address lines are shared by all components, there must be a means of controlling their use. Control signals transmit both command and liming information between system modules. Timing signals indicate the validity of data and address information.
- Command signals specify operations to be performed. Typical control lines include the following:

- Memory write: Causes data on the bus to be written into the addressed location.
- Memory read: Causes data from the addressed location to be placed on the bus.
- I/O write: Causes data on the bus to be output to the addressed I/O port.
- I/O read: Causes data from the addressed I/O port to be placed on the bus.
- Transfer ACK: Indicates that data have been accepted from or placed on the bus.
- Bus request: Indicates that a module needs to gain control of the bus.
- Bus grant: Indicates that a requesting module has been granted control of the bus.
- Interrupt request: Indicates that an interrupt is pending.
- Interrupt ACK: Acknowledges that the pending interrupt has been recognized.
- Clock: Used to synchronize operations.
- Reset: Initializes all modules.

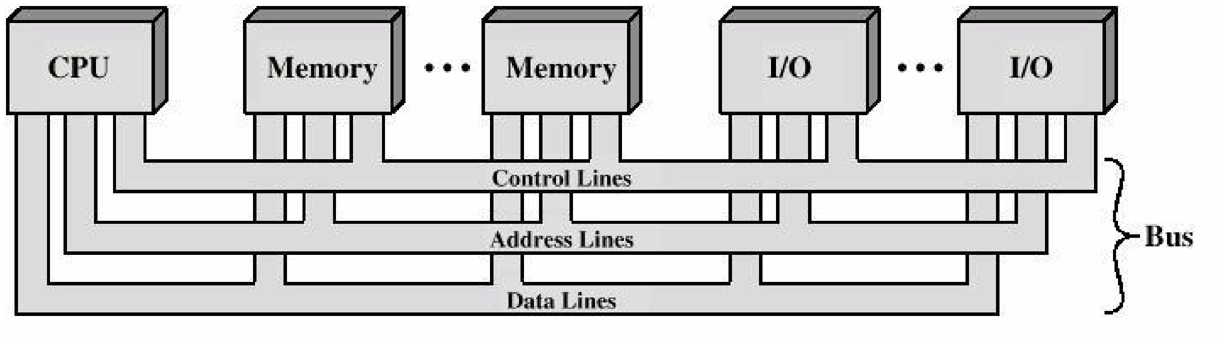


Figure 12 Bus Interconnection Scheme

Multiple-Bus Hierarchies

If a great number of devices are connected to the bus, performance will suffer. There are two main causes:

- In general, the more devices attached to the bus, the greater the bus length and hence the greater the propagation delay. This delay determines the time it takes for devices to coordinate the use of the bus. When control of the bus passes from one device to another frequently, these propagation delays can noticeably affect performance.
- The bus may become a bottleneck as the aggregate data transfer demand approaches the capacity of the bus. This problem can be countered to some extent by increasing the data rate that the bus can carry and by using wider buses (e.g., increasing the data bus from 32 to 64 bit). However, because the data rates generated by attached devices (e.g.. graphics and video controllers, network interfaces) are growing rapidly, this is a race that a single bus is ultimately destined to lose.

Accordingly, most computer systems use multiple buses, generally laid out in a hierarchy. A typical traditional structure is shown in Figure 13. There is a local bus that connects the processor to a cache memory and that may support one or more local devices. The cache memory controller connects the cache not only to this local bus, but to a system bus to which are attached all of the main memory modules.

It is possible to connected I/O controllers directly onto the system bus. A more efficient solution is to make use of one or more expansion buses for this purpose. An expansion bus interface buffers data transfers between the system bus and the I/O controllers on the expansion bus. This arrangement allows the system to support a wide variety of I/O devices and at the same time insulate memory-to-processor traffic from I/O traffic.

Traditional (ISA) (with cache):

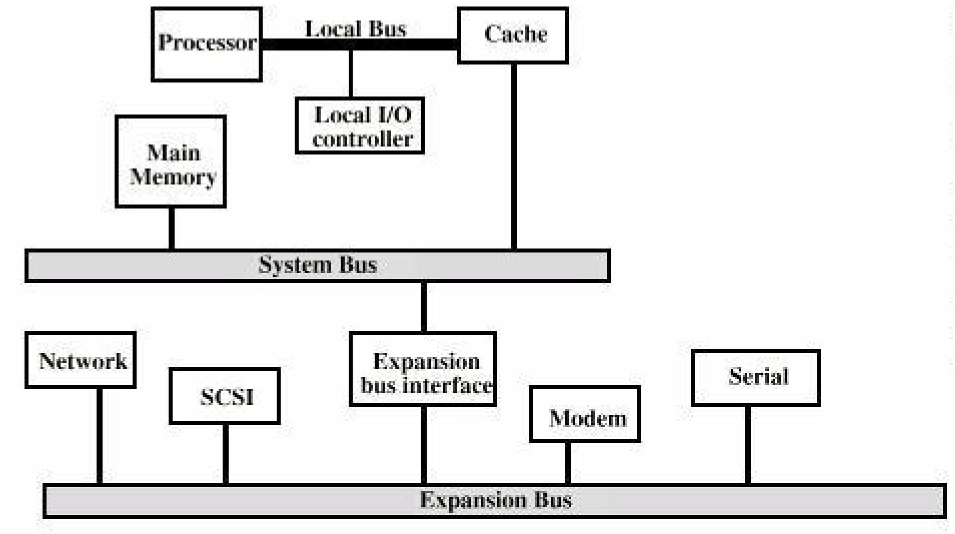


Figure 13 Traditional bus architecture

Elements of Bus Design

Type:

Dedicated

Multiplexed

Method of Arbitration:

Centralized

Distributed

Timing:

Synchronous

Asynchronous

Bus Width:

Address

Data

Data Transfer Type:

Read

Write

Read-modify-write

Read-after-write

Block

Module 3: Computer Arithmetic

1. The Arithmetic and Logic Unit

The ALU is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system —control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out. We have, in a sense, reached the core or essence of a computer when we consider the ALU. An ALU and, indeed, all electronic components in the computer arc based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

Figure 3.1 indicates, in general terms, how the ALU is interconnected with the rest of the processor. Data are presented to the ALU in registers, and the results of an operation are stored in registers. These registers are temporary storage locations within the processor that are connected by signal paths to the ALU. The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.

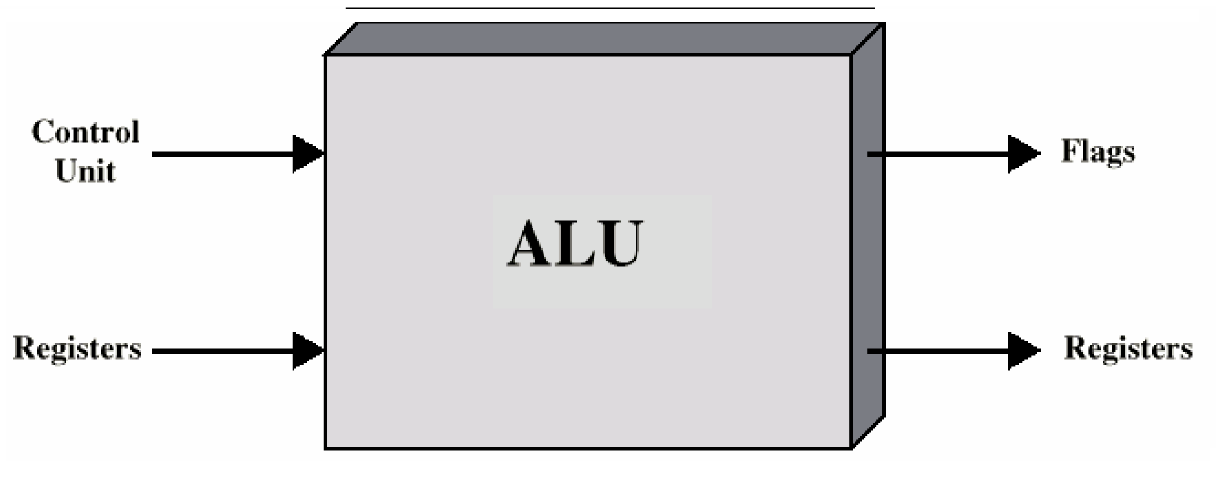


Figure 3.1 ALU Input and Output

2. Integer Representation

In the binary number system, arbitrary numbers can be represented with just the digits zero and one, the minus sign, and the period, or radix point.

For purposes of computer storage and processing, however, we do not have the benefit of minus signs and periods. Only binary digits (0 and 1) may be used lo represent numbers.

2.1 Unsigned Integer

If we are limited to nonnegative integers, the representation is straightforward.

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2}...a_1a_0$ is interpreted as an unsigned integer A, its value is
$A = a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + ... + a_1 2^1 + a_0 2^0$

$A = \sum_{i=0}^{n-1} a_i 2^i$

2.2 Signed Integer

Sign-magnitude representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive: if the sign bit is 1, the number is negative.

The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost n - 1 bits hold the magnitude of the integer.

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2}...a_1a_0$ represented A, its value is

$A = \sum_{i=0}^{n-2} a_i 2^i$ If $a_{n-1} = 0$

$A = -\sum_{i=0}^{n-2} a_i 2^i$ If $a_{n-1} = 1$

For example: +18= 0 0010010

- 18=1 0010010

There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. Another drawback is that there are two representations of 0 (e.g 0000 0000 and 1000 00000).

This is inconvenient, because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation.

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU. Instead, the most common scheme is twos complement representation.

Twos Complement Representation

- Ones Complement

The ones complement of a number is represented by flipping the number's bits one at a time. For example, the value 01001001 becomes 10110110.

Suppose you are working with B-bit numbers. Then the ones complement for the number N is 2^B-1 -N

- Twos complement

The twos complement of a number N in a B-bit system is 2^B-N.

There is a simple way to calculate a twos complement value: invert the number's bits and then add 1.

For a concrete example, consider the value 17 which is 00010001 in binary. Inverting gives 11101110. Adding one makes this 11101111.

- Twos Complement Representation

Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative. It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted.

Consider an n-bit integer. A, in twos complement representation. If A is positive, then the sign bit an-1 is zero. The remaining, bits represent the magnitude of the number in the same fashion as for sign magnitude:

$A = \sum_{i=0}^{n-2} a_i 2^i$ If $a_{n-1} = 0$

The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s. We can see that the range of positive integers that may he represented is from 0 (all of the magnitude bits are 0) through - 1 (all of the magnitude bits are 1). Any larger number would require more bits.

For a negative number A (A < 0), the sign bit, is one. The remaining n-1 bits can take on any one of $a_{n-1} 2^{n-1}$ values. Therefore, the range of negative integers that can be represented is from -1 to - 2^{n-1}

This is the convention used in twos complement representation, yielding the following expression for negative and positive numbers:

$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} a_i 2^i$

The range of A is from - 2^{n-1} to 2^{n-1} -1.

Example: Using 8 bit to represent

+50= 0011 0010

-70=1011 1010

Converting between Different Bit Lengths

The rule for twos complement integers is to move the sign hit to the new leftmost position and fill in with copies of the sign bit. For positive numbers, fill in with zeros, and for negative numbers, till in with ones.

For example:

+18 = 00010010

+18 = 00000000 00010010

-18 = 10010010

-18 = 11111111 10010010

3. Integer Arithmetic

3.1 Negation

In sign-magnitude representation, the rule for forming the negation of an integer is simple: Invert the sign bit.

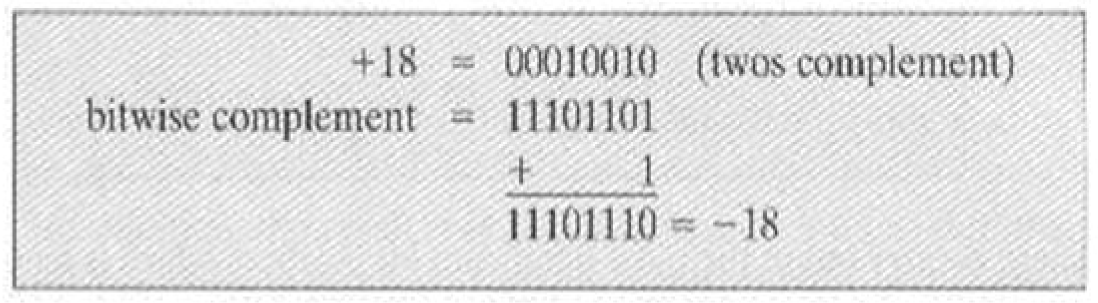
In twos complement representation, the negation of an integer can he formed with the following rules:

1. Take the Boolean complement of each bit of the integer (including the sign bit). That is. set each 1 to 0 and each 0 to 1.

2. Treating the result as an unsigned binary integer, add 1.

This two-step process is referred to as the twos complement operation, or the taking of the twos complement of an integer

Example:



```
               +18 = 00010010  (twos complement)
bitwise complement = 11101101
                     +      1
                     --------
                     11101110 = −18
```

Negation Special Case 1:

0 = 00000000

Bitwise not 11111111

Add 1 to LSB +1

Result 1 00000000

Overflow is ignored, so: - 0 = 0

Negation Special Case 2:

-128 = 10000000

bitwise not 01111111

Add 1 to LSB +1

Result 10000000

So: -(-128) = -128.

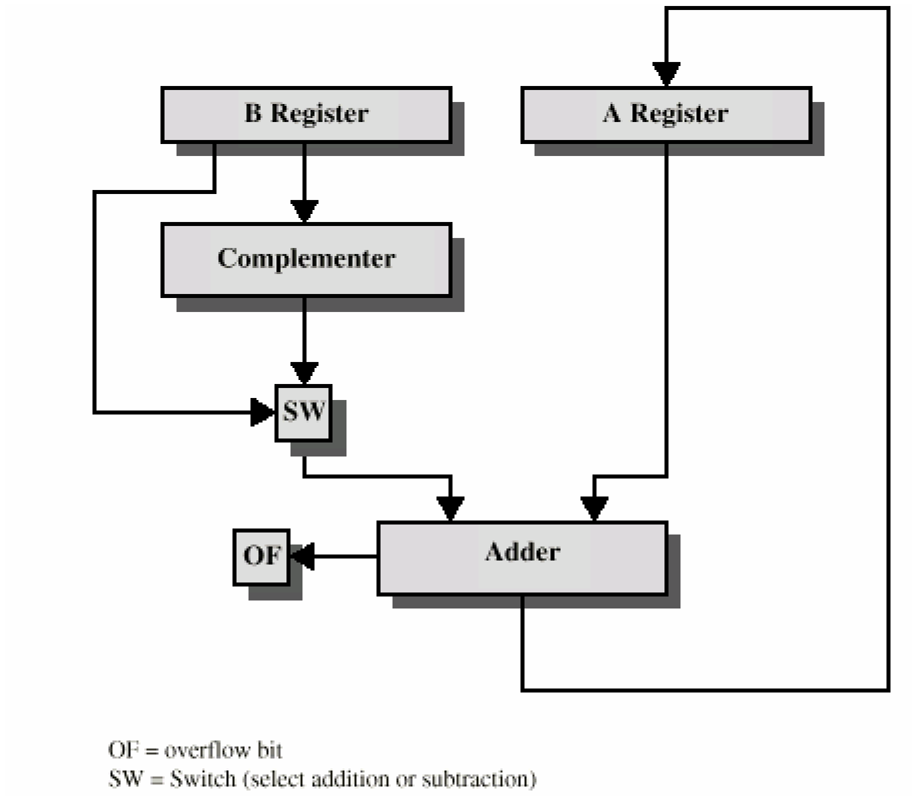
Because 128 is out of the range of signed 8bits numbers.

3.2 Addition and Subtraction:

Addition and Subtraction is done using following steps:

- Normal binary addition
- Monitor sign bit for overflow
- Take twos compliment of subtrahend and add to minuend ,i.e. a - b = a + (-b)

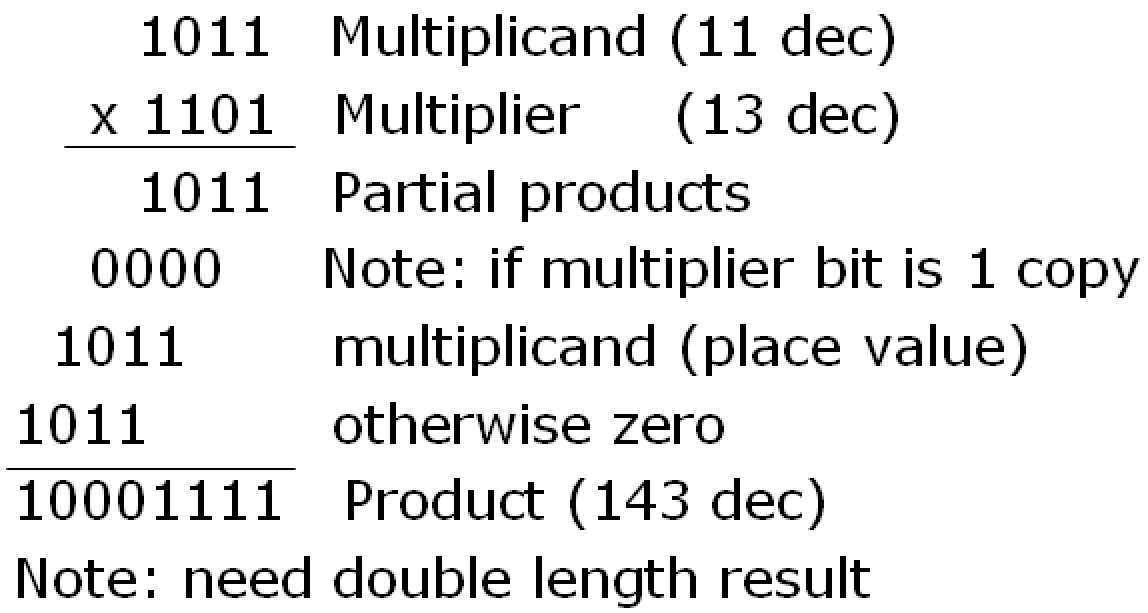
Hardware for Addition and Subtraction:



3.3 Multiplying positive numbers:

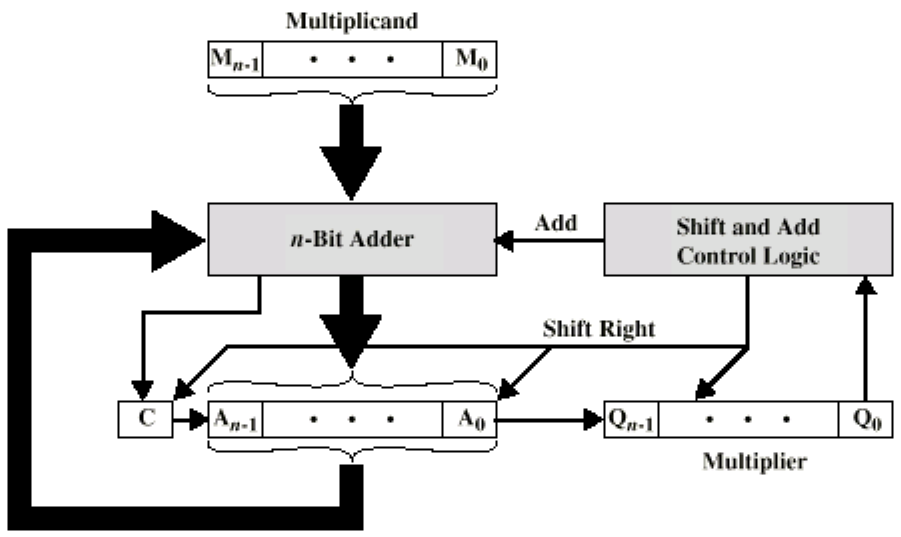
The multiplying is done using following steps:

- Work out partial product for each digit
- Take care with place value (column)
- Add partial products



```
      1011   Multiplicand (11 dec)
    x 1101   Multiplier     (13 dec)
   ---------
      1011   Partial products
     0000    Note: if multiplier bit is 1 copy
    1011     multiplicand (place value)
   1011      otherwise zero
  ----------
  10001111   Product (143 dec)
  Note: need double length result
```

Hardware Implementation of Unsigned Binary Multiplication:



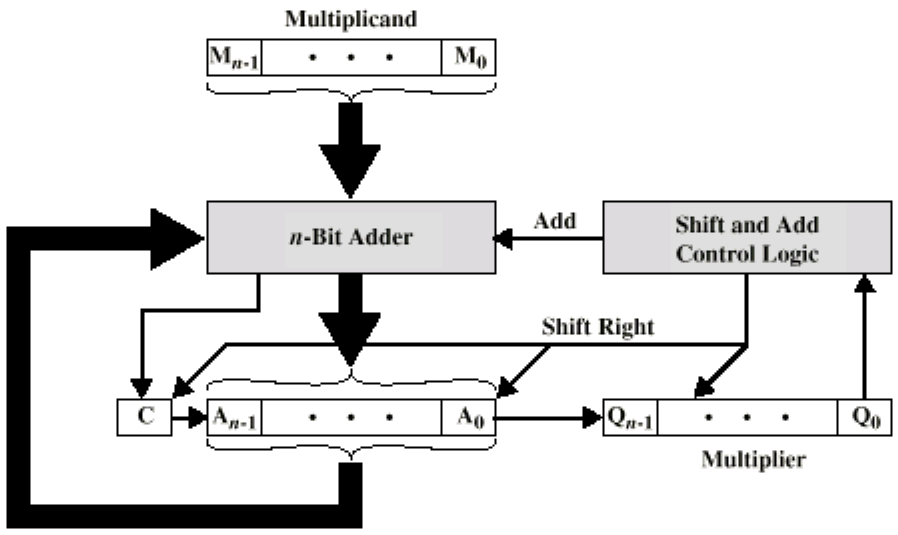
(a) Block Diagram

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 8.7 (product in A, Q)

Figure 8.8 Hardware Implementation of Unsigned Binary Multiplication

Execution of Example:



(a) Block Diagram

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 8.7 (product in A, Q)

Figure 8.8 Hardware Implementation of Unsigned Binary Multiplication

Flowchart for Unsigned Binary Multiplication:

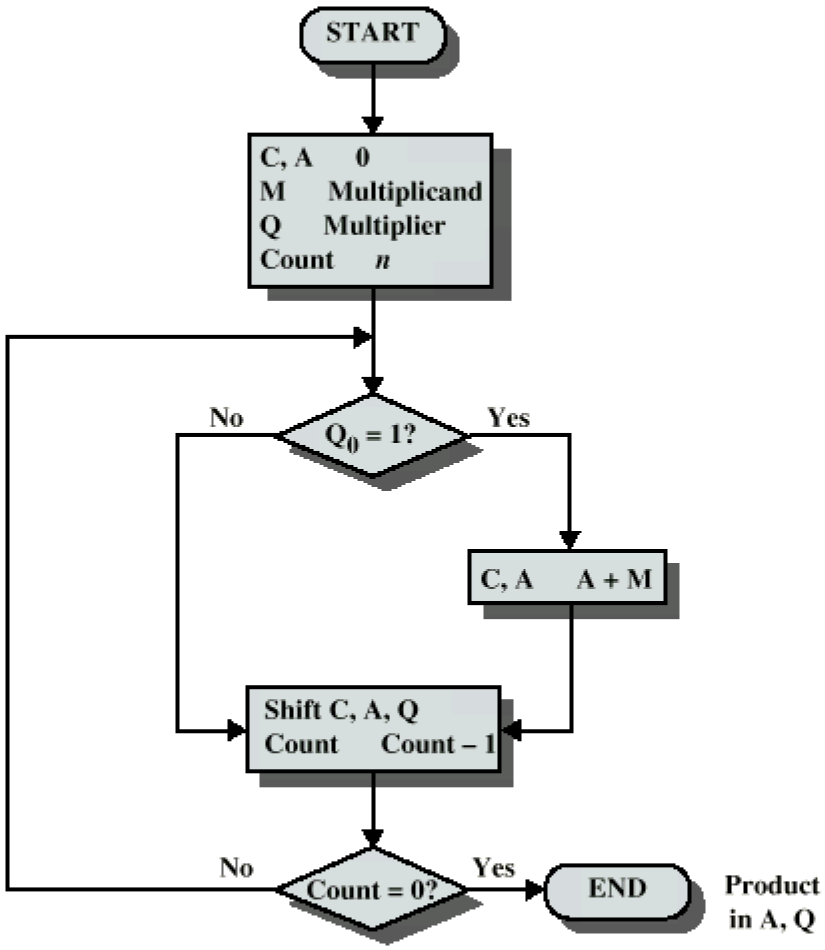


Figure 8.9 Flowchart for Unsigned Binary Multiplication

3.4 Multiplying Negative Numbers

Solution 1:

- Convert to positive if required
- Multiply as above
- If signs were different, negate answer

Solution 2:

- Booth’s algorithm:

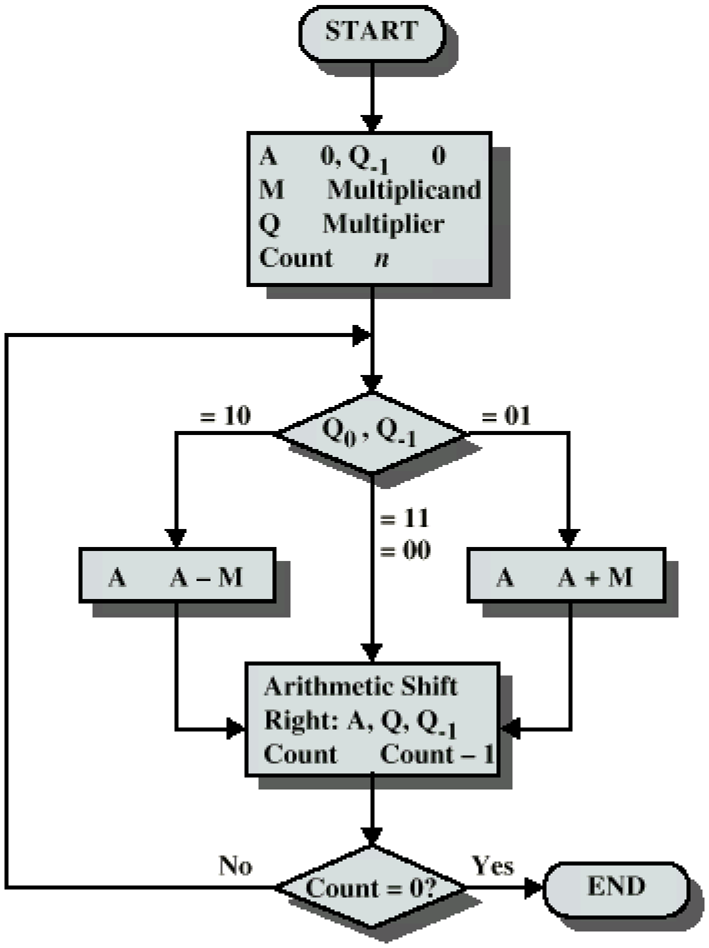
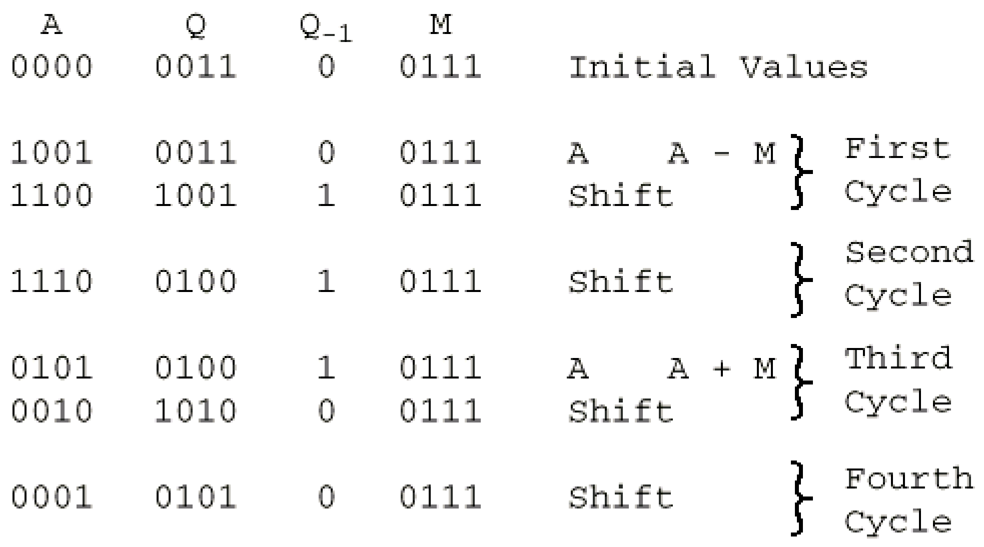


Figure 8.12 Booth's Algorithm for Twos Complement Multiplication

Example of Booth’s Algorithm:



A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Figure 8.13 Example of Booth's Algorithm

3.5 Division:

- More complex than multiplication
- Negative numbers are really bad!
- Based on long division
- (for more detail, reference to Computer Organization and Architecture, William Stalling)

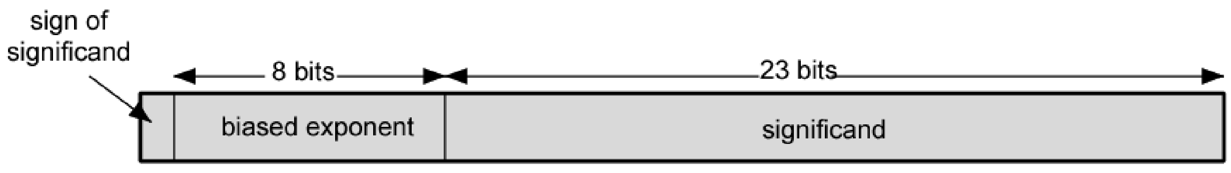
4. Floating-Point Representation

4.1 Principles

We can represent a real number in the form

$\pm S \times B^{\pm E}$

This number can be stored in a binary word with three fields:



(a) Format

1.1010001 ´ 2^{10100} = 0 10010011 10100010000000000000000 = 1.6328125 ´ 2^{20}

-1.1010001 ´ 2^{10100} = 1 10010011 10100010000000000000000 = -1.6328125 ´ 2^{20}

1.1010001 ´ 2^{-10100} = 0 01101011 10100010000000000000000 = 1.6328125 ´ 2^{-20}

-1.1010001 ´ 2^{-10100} = 1 01101011 10100010000000000000000 = -1.6328125 ´ 2^{-20}

(b) Examples

Figure 9.18 Typical 32-Bit Floating-Point Format

- Sign: plus or minus
- Significant: S
- Exponent: E.

(A fixed value, called the bias, is subtracted from the biased exponent field to get the true exponent value (E). Typically, the bias equal $2^{k-1} - 1$, where k is the number of bits in the binary exponent)

- The base B is implicit and need not be stored because it is the same for all numbers.

4.2 IEEE Standard for Binary Floating-Point Representation

The most important floating-point representation is defined in IEEE Standard 754 [EEE8]. This standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs. The standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors.

The IEEE standard defines both a 32-bit (Single-precision) and a 64-bit (Double-precision) double format with 8-bit and 11-bit exponents, respectively. Binary floating-point numbers are stored in a form where the MSB is the sign bit, exponent is the biased exponent, and "fraction" is the significand. The implied base (B) is 2.

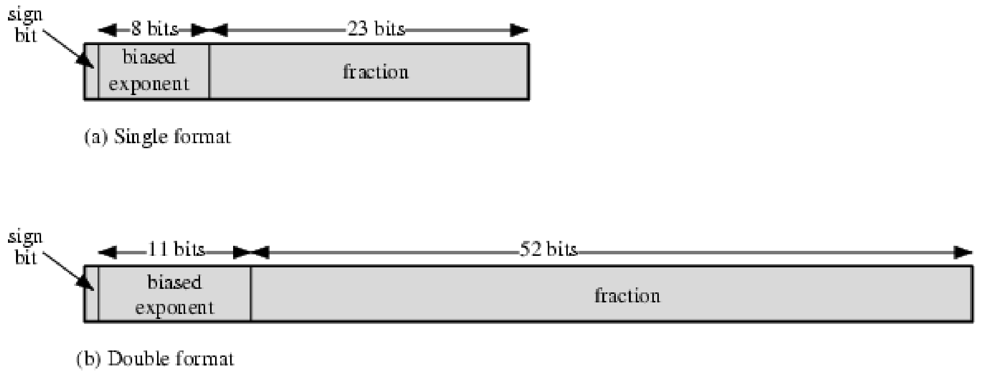


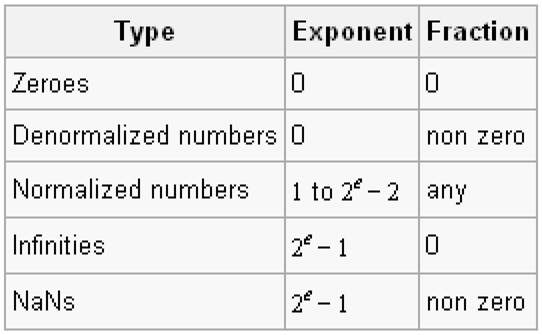
Figure 9.21 IEEE 754 Formats

Not all bit patterns in the IEEE formats are interpreted in die usual way; instead, some bit patterns are used to represent special values. Three special cases arise:

1. if exponent is 0 and fraction is 0, the number is ±0 (depending on the sign bit)
2. if exponent = 2^e-1 and fraction is 0, the number is ±infinity (again depending on the sign bit), and

3. if exponent = 2^e-1 and fraction is not 0, the number being represented is not a number (NaN).

This can be summarized as:



Type	Exponent	Fraction
Zeroes	0	0
Denormalized numbers	0	non zero
Normalized numbers	1 to $2^e - 2$	any
Infinities	$2^e - 1$	0
NaNs	$2^e - 1$	non zero

Single-precision 32 bit

A single-precision binary floating-point number is stored in 32 bits.

The number has value v:

v = s × 2^e× m

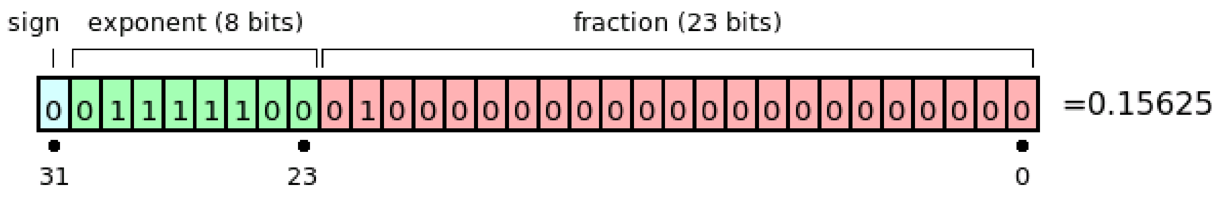
Where

s = +1 (positive numbers) when the sign bit is 0

s = −1 (negative numbers) when the sign bit is 1

e = Exp − 127 (in other words the exponent is stored with 127 added to it, also called "biased with 127")

m = 1.fraction in binary (that is, the significand is the binary number 1 followed by the radix point followed by the binary bits of the fraction). Therefore, 1 ≤ m < 2.



In the example shown above:

S=1

E= 011111100(2) -127 = -3

M=1.01 (in binary, which is 1.25 in decimal).

The represented number is: +1.25 × 2−3 = +0.15625.

5. Floating-Point Arithmetic

The basic operations for floating-point $X1 = M1 * R^{E1}$ and $X2 = M2 * R^{E2}$

- $X1 \pm X2 = (M1 * R^{E1-E2})R^{E2}$ (assume E1 E2)
- $X1 * X2 = (M1 * M2)R^{E1+E2}$
- $X1 / X2 = (M1 / M2)R^{E1-E2}$

For addition and subtraction, it is necessary lo ensure that both operands have the same exponent value. I his may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are more straightforward.

A floating-point operation may produce one of these conditions:

- Exponent overflow: A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as
- Exponent underflow: A negative exponent is less than the minimum possible exponent value (e.g.. -200 is less than -127). This means that the number is too small to be represented, and it may be reported as 0.

- Significand underflow: In the process of aligning significands, digits may flow off the right end of the significand. Some form of rounding is required.
- Significand overflow: The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment.

Module 4: Instruction Set: Characteristics and Functions

1. Machine Instruction Characteristics

1.1 What is an Instruction Set?

From the designer's point of view, the machine instruction set provides the functional requirements for the CPU: Implementing the CPU is a task that in large part involves implementing the machine instruction set.

From the user's side, the user who chooses to program in machine language (actually, in assembly language) becomes aware of the register and memory structure, the types of data directly supported by the machine, and the functioning of the ALU.

1.2 Elements of an Instruction

Each instruction must have elements that contain the information required by the CPU for execution. These elements are as follows

- Operation code: Specifies the operation to be performed (e.g.. ADD, I/O). The operation is specified by a binary code, known as the operation code, or opcode.
- Source operand reference: The operation may involve one or more source operands, that is, operands that are inputs for the operation.
- Result operand reference: The operation may produce a result.
- Next instruction reference: This tells the CPU where to fetch the next instruction after the execution of this instruction is complete.

The next instruction to be fetched is located in main memory or, in the case of a virtual memory system, in either main memory or secondary memory (disk). In most cases, the next instruction to be fetched immediately follows the current instruction. In those cases, there is no explicit reference to the next instruction. Source and result operands can be in one of three areas:

- Main or virtual memory: As with next instruction references, the main or virtual memory address must be supplied.
- CPU register: With rare exceptions, a CPU contains one or more registers that may be referenced by machine instructions. If only one register exists, reference to it may be implicit. If more than one register exists, then each register is assigned a unique number, and the instruction must contain the number of the desired register.
- I/O device: The instruction must specify (he I/O module and device for the operation. If memory-mapped I/O is used, this is just another main or virtual memory address.

Instruction Cycle State Diagram

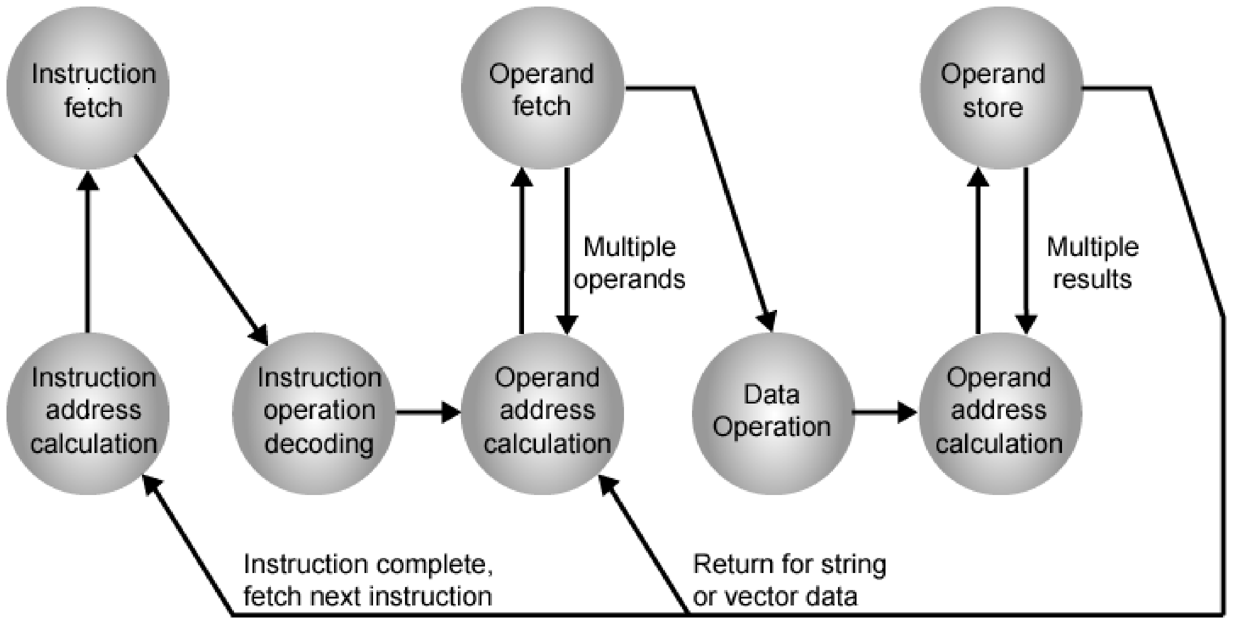

Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data

Figure 10.1 Instruction Cycle State Diagram

Instruction Representation

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the instruction. During instruction execution, an instruction is read into an instruction register (IR) in the CPU. The CPU must be able to extract the data from the various instruction fields to perform the required operation.

It is difficult for both the programmer and the reader of textbooks to deal with binary representations of machine instructions. Thus, it has become common practice to use a symbolic representation of machine instructions. Opcodes are represented by abbreviations, called mnemonics, that indicate the operation. Common examples include

ADD	Add
SUB	Subtract
MPY	Multiply
DIV	Divide
LOAD	Load data from memory
STOR	Store data to memory

Operands are also represented symbolically. For example, the instruction

ADD R, Y

may mean add the value contained in data location Y to the contents of register R. In this example. Y refers to the address of a location in memory, and R refers to a particular register. Note that the operation is performed on the contents of a location, not on its address.

Simple Instruction Format

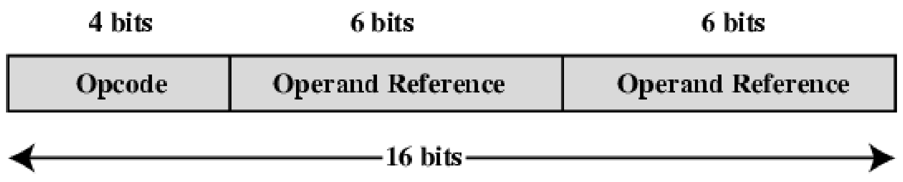


Figure 10.2 A Simple Instruction Format

1.3 Instruction Types

Consider a high-level language instruction that could be expressed in a language such as BASIC or FORTRAN. For example,

X = X+Y

This statement instructs the computer lo add the value stored in Y to the value Stored in X and put the result in X. How might this be accomplished with machine instructions? Let us assume that the variables X and Y correspond lo locations 513 and 514. If we assume a simple set of machine instructions, this operation could be accomplished with three instructions:

1. Load a register with the contents of memory location 513.

2. Add the contents of memory location 514 to the register.

3. Store the contents of the register in memory location 513.

As can be seen, the single BASIC instruction may require three machine instructions. This is typical of the relationship between a high-level language and a machine language. A high-level language expresses operations in a concise algebraic form, using variables. A machine language expresses operations in a basic form involving the movement of data to or from registers.

With this simple example to guide us, let us consider the types of instructions that must be included in a practical computer. A computer should have a set of instructions that allows the user to formulate any data processing task. Another way to view it is to consider the capabilities of a high-level programming language. Any program written in a high-level language must be translated into machine language to be executed. Thus, the set of machine instructions must be sufficient to express any of the instructions from a high-level language. With this in mind we can cate-gorize instruction types as follows:

- Data processing: Arithmetic and logic instructions
- Data storage: Memory instructions
- Data movement: I/O instructions
- Control: Test and branch instructions

1.4 Number of Addresses

What is the maximum number of addresses one might need in an instruction? Evidently, arithmetic and logic instructions will require the most operands. Virtually all arithmetic and logic operations are either unary (one operand) or binary (two operands). Thus, we would need a maximum of two addresses to reference operands. The result of an operation must be stored, suggesting a third address. Finally, after completion of an instruction, the next instruction must be fetched, and its address is needed.

This line of reasoning suggests that an instruction could plausibly be required to contain four address references: two operands, one result and the address of the next instruction. In practice, four-address instructions are extremely rare. Most instructions have one, two, or three operand addresses, with the address of the next instruction being implicit (obtained from the program counter).

- Three addresses:
- Operand 1, Operand 2, Result

Example: a = b + c

- Three-address instruction formats are not common, because they require a relatively long instruction format to hold the three address references.
- Two addresses:
- One address doubles as operand and result

Example: a = a + b

- The two-address formal reduces the space requirement but also introduces some awkwardness. To avoid altering the value of an operand, a MOVE instruction is used to move one of the values to a result or temporary location before performing the operation.
- One addresses:

- a second address must be implicit. This was common in earlier machines, with the implied address being a CPU register known as the accumulator. or AC. The accumulator contains one of the operands and is used to store the result.

- Zero addresses

- Zero-address instructions are applicable to a special memory organization, called a Stack. A stack is a last-in-first-out set of locations.

How Many Addresses?

The number of addresses per instruction is a basic design decision.

Fewer addresses:

- Fewer addresses per instruction result in more primitive instructions, which requires a less complex CPU.
- It also results in instructions of shorter length. On the other hand, programs contain more total instructions, which in general results in longer execution times and longer, more complex programs

Multiple-address instructions:

- With multiple-address instructions, it is common to have multiple general-purpose registers. This allows some operations to be performed solely on registers.
- Because register references are faster than memory references, this speeds up execution.

1.5 Design Decisions

One of the most interesting and most analyzed, aspects of computer design is instruction set design. The design of an instruction set is very complex, because it affects so many aspects of the computer system. The instruction set defines many of the functions performed by the CPU and thus has a significant effect on the implementation of the CPU. The instruction set is the programmer's means of controlling the CPU. Thus, programmer requirements must be considered in designing the instruction set. The most important design issues include the following:

- Operation repertoire: How many and which operations to provide, and how complex operations should be
- Data types: The various types of data upon which operations are performed
- Instruction format: Instruction length (in bits), number of addresses, size of various fields, and so on.
- Registers: Number of CPU registers that can be referenced by instructions, and their use.
- Addressing: The mode or modes by which the address of an operand is specified

2. Types of Operands

Machine instructions operate on data. The most important general categories of data are:

- Addresses
- Numbers
- Characters
- Logical data

2.1 Numbers

All machine languages include numeric data types. Even in nonnumeric data processing, there is a need for numbers to act as counters, field widths, and so forth. An important distinction between numbers used in ordinary mathematics and numbers stored in a computer is that the latter are limited. Thus, the programmer is faced with understanding the consequences of rounding, overflow and underflow.

Three types of numerical data are common in computers:

- Integer or fixed point
- Floating point
- Decimal

2.2 Characters

A common form of data is text or character strings. While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems. Such systems are designed for binary data. Thus, a number of codes have been devised by which characters are represented by a sequence of bits. Perhaps the earliest common example of this is the Morse code. Today, the most commonly used character code in the International Reference Alphabet (IRA), referred to in the United States as the American Standard Code for Information Interchange (ASCII). IRA is also widely used outside the United States. Each character in this code is represented by a unique 7-bit pattern, thus, 128 different characters can be represented. This is a larger number than is necessary to represent printable characters, and some of the patterns represent control characters. Some of these control characters have to do with controlling the printing of characters on a page. Others are concerned with communications procedures. IRA-encoded characters are almost always stored and transmitted using 8 bits per character. The eighth bit may be set to 0 or used as a parity bit for error detection. In the latter case, the bit is set such that the total number of binary 1s in each octet is always odd (odd parity) or always even (even parity).

Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC). EBCDIC is used on IBM S/390 machines. It is an 8-bit code. As with IRA, EBCDIC is compatible with packed decimal. In the case of EBCDIC, the codes 11110000 through 11111001 represent the digits 0 through 9.

2.3 Logical Data

Normally, each word or other addressable unit (byte, half-word, and soon) is treated as a single unit of data. It is sometimes useful, however, to consider an n-bit unit as consisting 1-bit items of data, each item having the value 0 or I. When data are viewed this way, they are considered to be logic data.

There are two advantages to the bit-oriented view:

- First, we may sometimes wish to store an array of Boolean or binary data items, in which each item can take on only the values I (true) and II (fake). With logical data, memory can be used most efficiently for this storage.
- Second, there are occasions when we wish to manipulate the bits of a data item.

3. Types of Operations

The number of different opcodes varies widely from machine to machine. However, the same general types of operations are found on all machines. A useful and typical categorization is the following:

- Data transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System control
- Transfer of control

3.1 Data transfer

The most fundamental type of machine instruction is the data transfer instruction. The data transfer instruction must specify several things.

- The location of the source and destination operands must be specified. Each location could be memory. a register, or the lop of the stack.
- The length of data to be transferred must be indicated.
- As with all instructions with operands, the mode of addressing for each operand must be specified.

In term of CPU action, data transfer operations are perhaps the simplest type. If both source and destination are registers, then the CPU simply causes data to be transferred from one register to another; this is an operation internal to the CPU. If one or both operands are in memory, then (he CPU must perform some or all of following actions:

1. Calculate the memory address, based on the address mode

2. If the address refers to virtual memory, translate from virtual to actual memory address.

3. Determine whether the addressed item is in cache.

4. If not, issue a command lo the memory module.

Example:

Operation mnemonic	Name	Number of bits transferred	Description
L	Load	32	Transfer from memory in register
LH	Load half-word	16	Transler from memory to register

ST	Store	32	Transfer from register to memory
STH	Store half-word	16	Transfer from register to memory

3.2 Arithmetic

Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. These are invariably provided for signed integer (fixed-point) numbers, Often they are also provided for floating-point and packed decimal numbers.

Other possible operations include a variety of single-operand instructions: for example.

• Absolute: Take the absolute value of the operand.

• Negate: Negate the Operand.

• Increment.: Add 1 to the operand.

• Decrement: Subtract 1 from the operand

3.3 Logical

Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable units, often referred to as "bit twiddling." They are based upon Boolean operations.

Some of the basic logical operations that can be performed on Boolean or binary data are AND, OR, NOT, XOR, …

These logical operations can be applied bitwise to n-bit logical data units. Thus, if two registers contain the data

(R1) - 10100101 (R2) - 00001111

then

(R1) AND (R2) – 00000101

In addition lo bitwise logical operations, most machines provide a variety of shifting and rotating functions such as shift left, shift right, right rotate, left rotate…

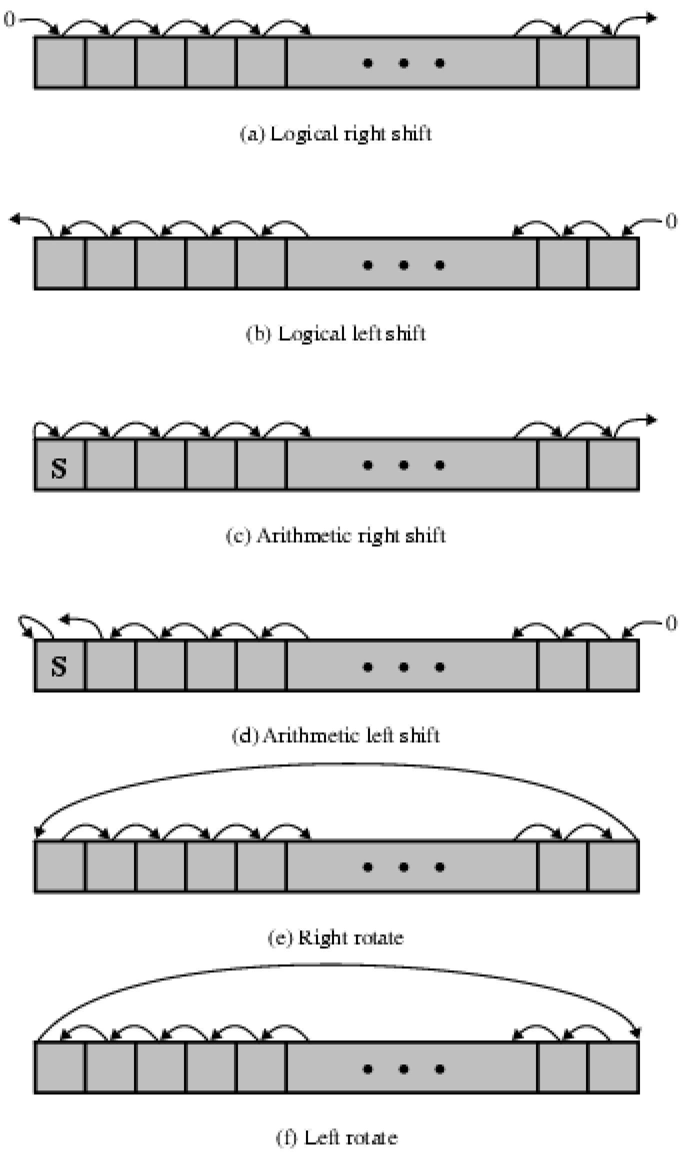


Figure 10.5 Shift and Rotate Operations

3.4 Conversion

Conversion instructions are those that change the format or operate on the format of data. An example is converting from decimal to binary.

3.5 Input/Output

As we saw, there are a variety of approaches taken, including isolated programmed IO, memory-mapped programmed I/O, DMA, and the use of an I/O processor. Many implementations provide only a few I/O instructions, with the specific actions specified by parameters, codes, or command words.

3.6 System Controls

System control instructions are those that can he executed only while the processor is in a certain privileged state or is executing a program in a special privileged area of memory, typically, these instructions are reserved for the use of the operating system.

Some examples of system control operations are as follows. A system control instruction may read or alter a control register. Another example is an instruction to read or modify a storage protection key, such us is used in the S/390 memory system. Another example is access to process control blocks in a multiprogramming system.

3.7 Transfer of control

For all of the operation types discussed so far. The next instruction to be performed is the one that immediately follows, in memory, the current instruction. However, a significant fraction of the instructions in any program have as their function changing the sequence of instruction execution. For these instructions, the operation performed by the CPU is to update the program counter to contain the address of some instruction in memory.

There are a number of reasons why transfer-of-control operations are required. Among the most important are the following:

1. In the practical use of computers, it is essential to be able to execute each instruction more than once and perhaps many thousands of times. It may require thousands or perhaps millions of instructions to implement an application. This would be unthinkable if each instruction had to be written out separately. If a table or a list of items is to be processed, a program loop is needed. One sequence of instructions is executed repeatedly to process all the data.

2. Virtually all programs involve some decision making. We would like the computer to do one thing if one condition holds, and another thing if another condition holds. For example, a sequence of instructions computes the square root of a number. At the start of the sequence, the sign of the number is tested, If the number is negative, the computation is not performed, but an error condition is reported.

3. To compose correctly a large or even medium-size computer program is an exceedingly difficult task. It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time.

We now turn to a discussion of the most common transfer-of-control operations found in instruction sets: branch, skip, and procedure call.

Branch instruction

A branch instruction, also called a jump instruction, has as one of its operands the address of the next instruction to be executed. Most often, the instruction is a conditional branch instruction. That is, the branch is made (update program counter to equal address specified in operand) only if a certain condition is met. Otherwise, the next instruction in sequence is executed (increment program counter as usual).

Skip instructions

Another common form of transfer-of-control instruction is the skip instruction. The skip instruction includes an implied address. Typically, the skip implies that one instruction be skipped; thus, the implied address equals the address of the next instruction plus one instruction-length.

Procedure call instructions

Perhaps the most important innovation in the development of programming languages is the procedure, a procedure is a self-contained computer program that is incorporated into a larger program. At any point in the program the procedure may he invoked, or called. The processor is instructed to go and execute the entire procedure and then return to the point from which the call took place.

The two principal reasons for the use of procedures are economy and modularity. A procedure allows the same piece of code to be used many times. This is important for economy in programming effort and for making the most efficient use of storage space in the system (the program must be stored). Procedures also allow large programming tasks to be subdivided into smaller units. This use of modularity greatly eases the programming task.

The procedure mechanism involves two basic instructions: a call instruction that branches from the present location to the procedure, and a return instruction that returns from the procedure to the place from which it was called. Both of these are forms of branching instructions.

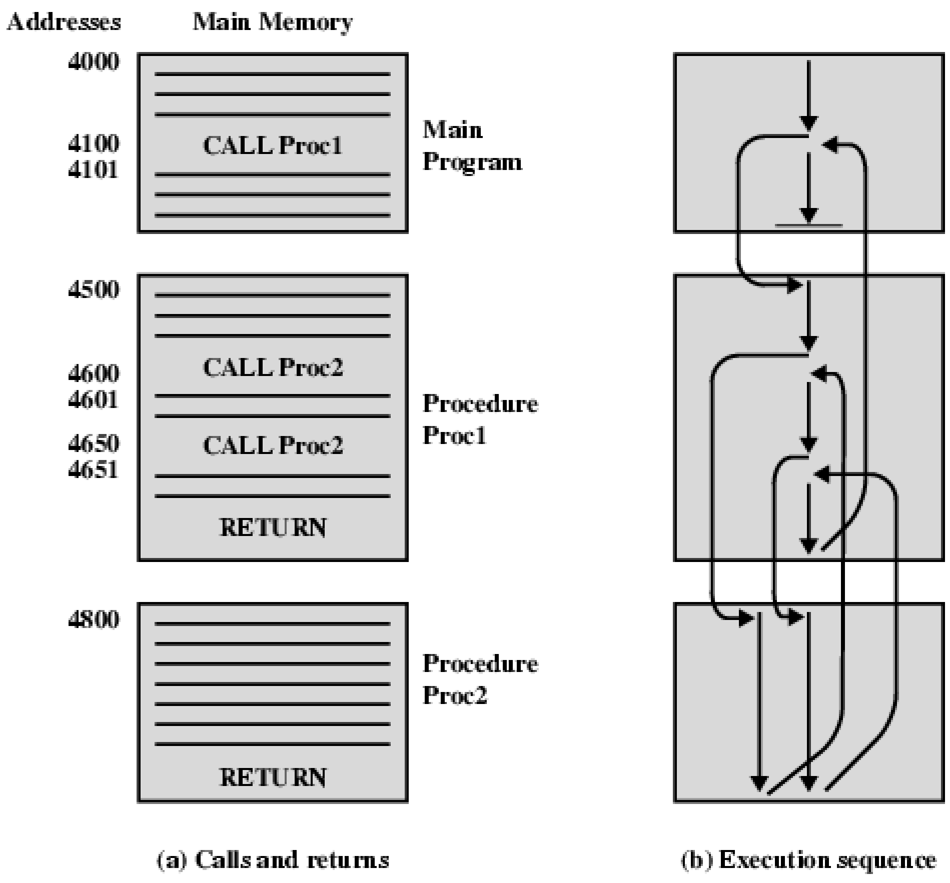


Figure 10.7 Nested Procedures

The above figure illustrates the use of procedures to construct a program. In this example, there is a main program starting at location 4000. This program includes a call to procedure PROC1, starting at location 4500. When this call instruction is encountered, the CPU suspends execution of the main program and begins execution of PROC1 by fetching the next instruction from location 4500. Within PROC1, there are two calls to PR0C2 at location 4800. In each case, the execution of PROC1 is suspended and PROC2 is executed. The RETURN statement causes the CPU to go back to the calling program and continue execution at the instruction after the corresponding CALL instruction. This behavior is illustrated in the right of this figure.

Several points are worth noting:

1. A procedure can be called from more than one location.

2. A procedure call can appear in a procedure. This allows the nesting of procedures to an arbitrary depth.

3. Each procedure call is matched by a return in the called program.

Because we would like to be able to call a procedure from a variety of points, the CPU must somehow save the return address so that the return can take place appropriately. There are three common places for storing the return address:

• Register

• Start of called procedure

• Top of stack

4. Addressing Modes

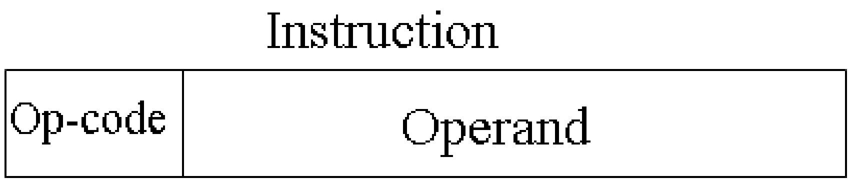
The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references and/or the complexity of address calculation, on the other. In this section, we examine the most common addressing techniques:

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement

4.1 Immediate Addressing

The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction:

- Operand is part of instruction
- Operand = address field
- e.g. ADD 5 ;Add 5 to contents of accumulator ;5 is operand



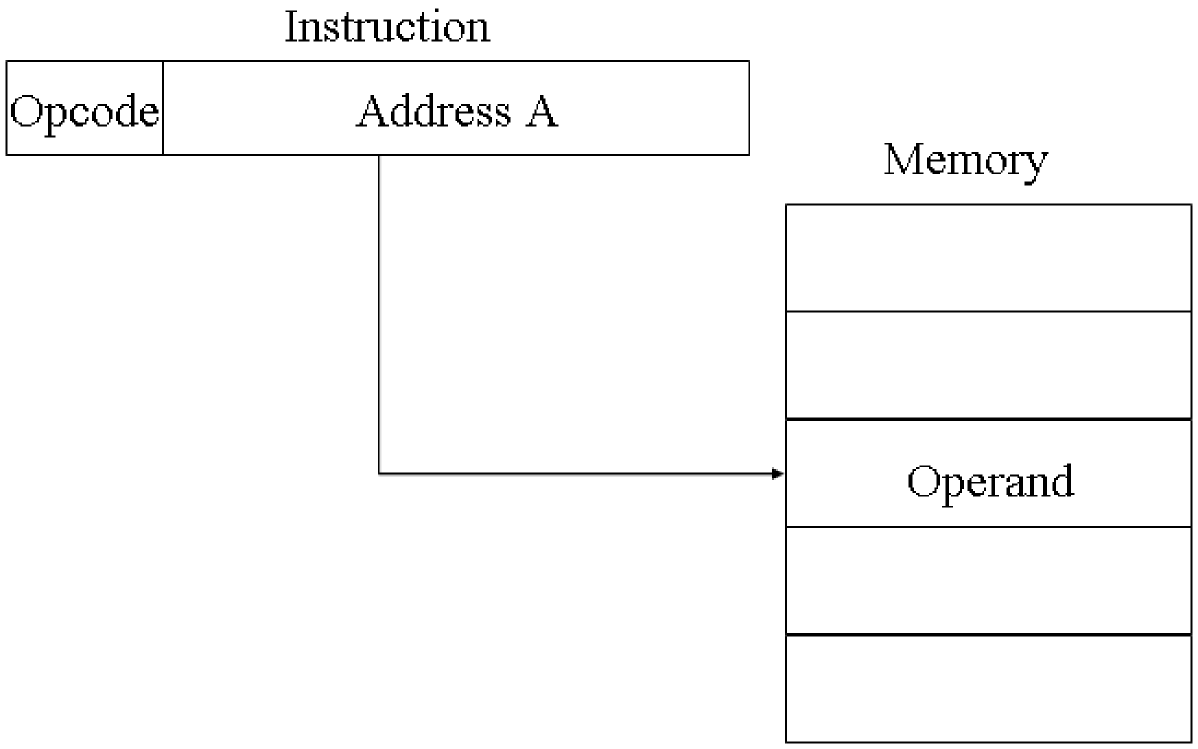
The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle.

The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

4.2 Direct Addressing

A very simple form of addressing is direct addressing, in which:

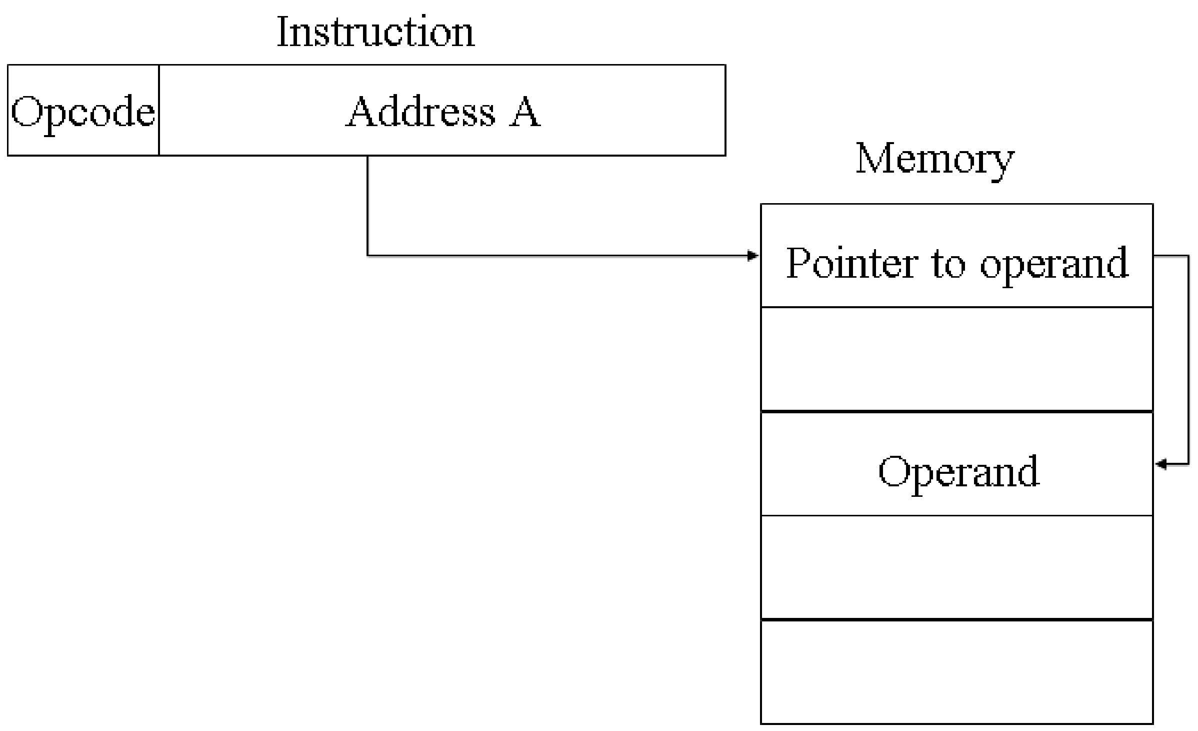
- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g. ADD A ;Add contents of cell A to accumulator



The technique was common in earlier generations of computers but is not common on contemporary architectures. It requires only one memory reference and no special calculation. The obvious limitation is that it provides only a limited address space.

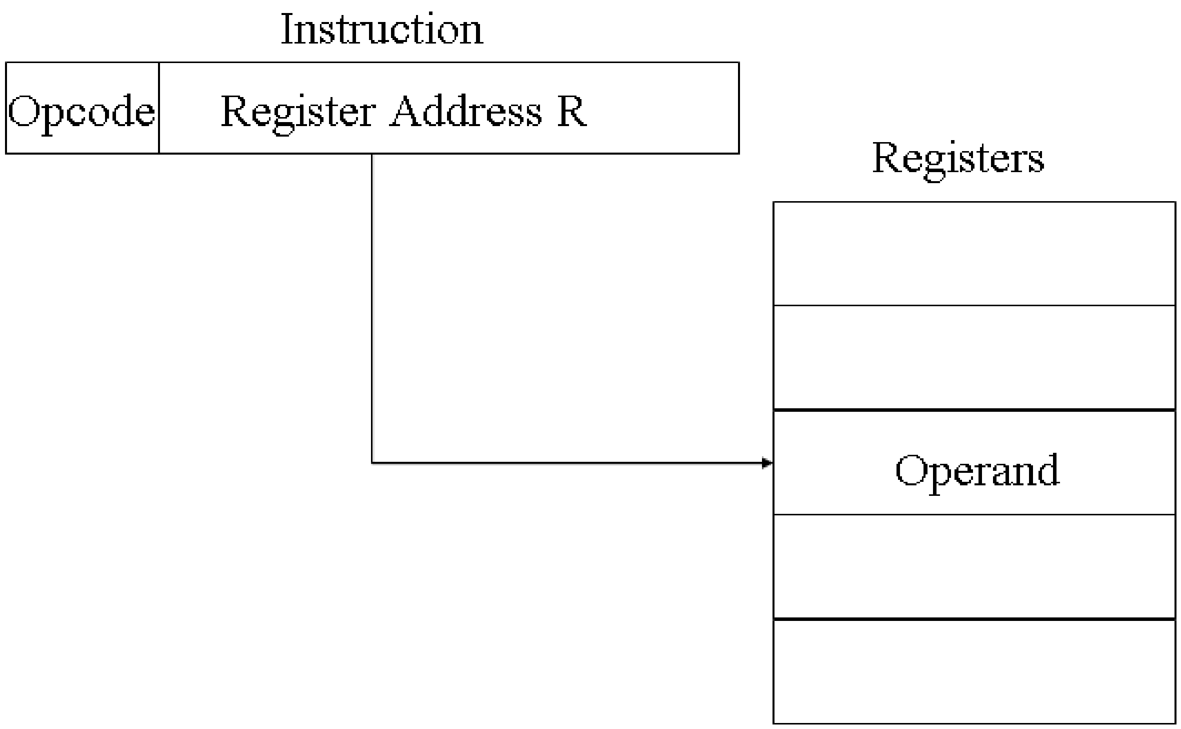
4.3 Indirect Addressing

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as indirect addressing.



4.4 Register Addressing

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address.



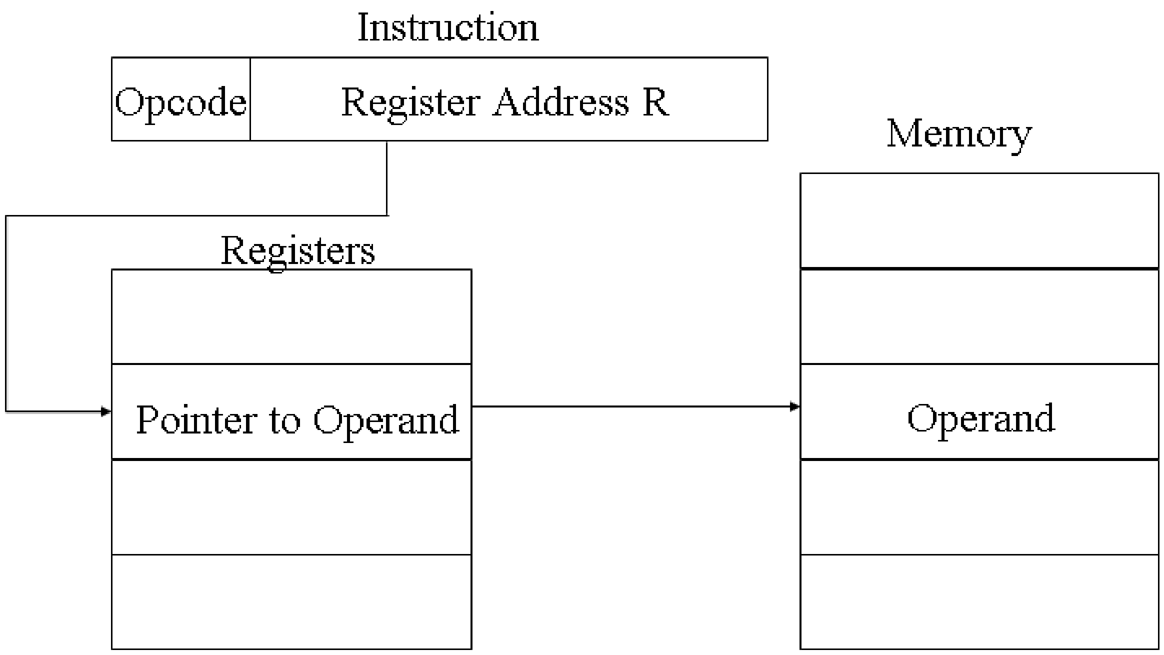
The advantages of register addressing are that :

- Only a small address field is needed in the instruction
- No memory 'references are required, faster instruction fetch

The disadvantage of register addressing is that the address space is very limited.

4.5 Register Indirect Addressing

Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register. Thus, for register indirect address: Operand is in memory cell pointed to by contents of register.

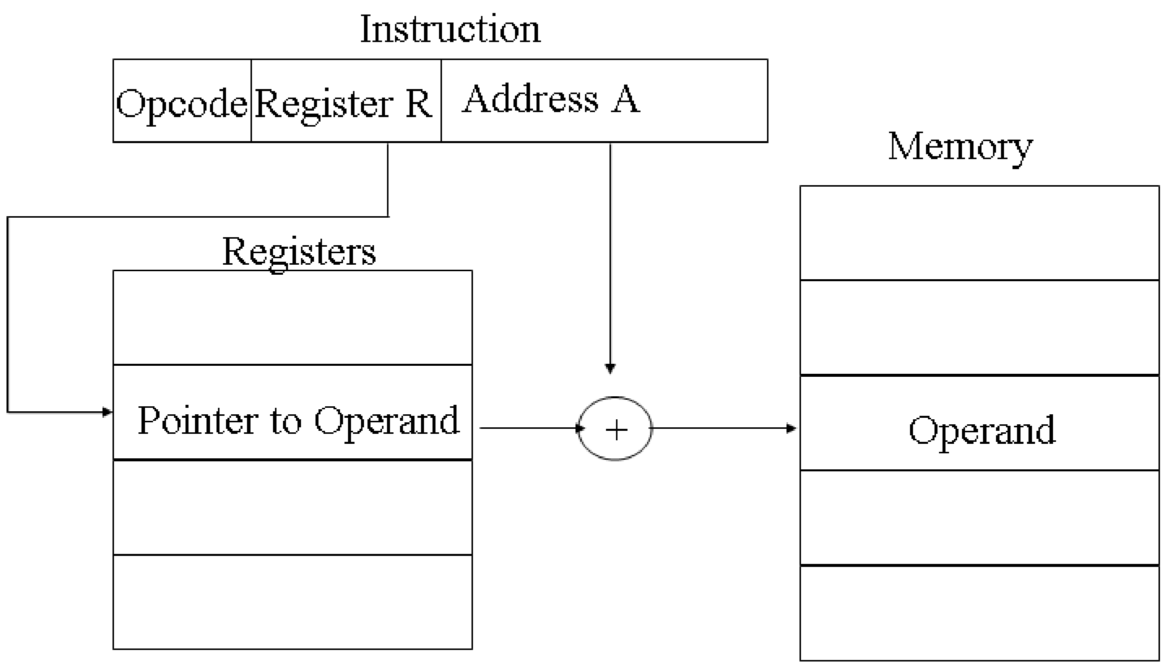


The advantages and limitations of register indirect addressing are basically the same as for indirect addressing. In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address. In addition, register indirect addressing uses one less memory reference than indirect addressing.

4.6 Displacement Addressing

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context of its use but the basic mechanism is the same. We will refer to this as displacement addressing, address field hold two values:

- A = base value
- R = register that holds displacement


Instruction
Opcode
Register R
Address A
Registers
Pointer to Operand
+
Memory
Operand

Module 5: CPU Structure and Functions

This module present the processor organization. We will begin with a summary of processor organization. Registers, which form the internal memory of the processor, are then analyzed. We are then in a position to return to the discussion of the instruction cycle. A description of the instruction cycle and a common technique known as instruction pipelining completes our description. Understanding the organization of the CPU of computer

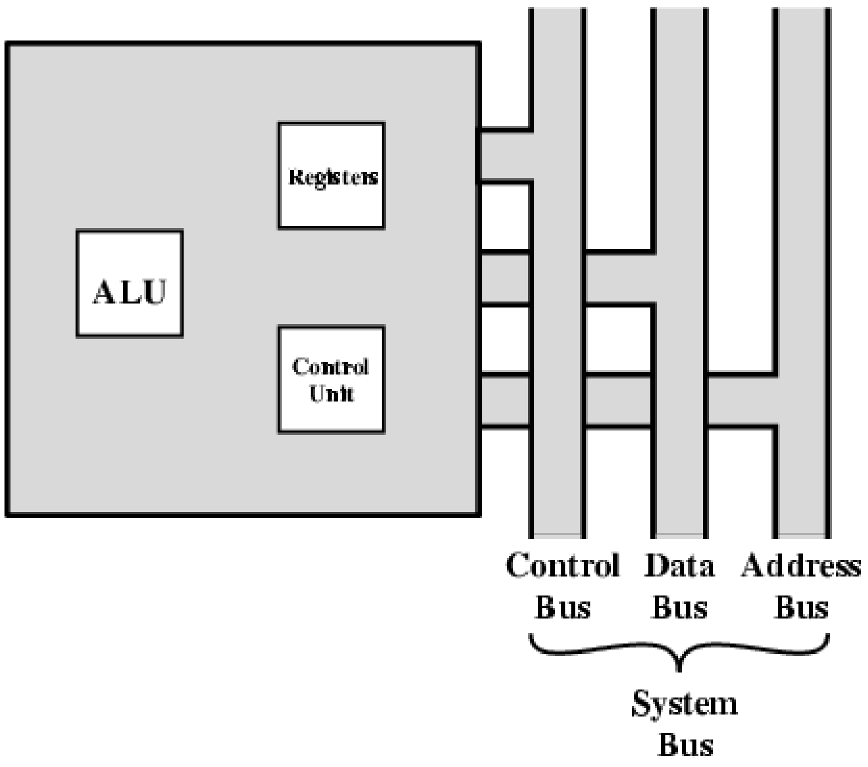
Processor Organization

To understand the organization of the CPU, let us consider the requirements placed on the CPU, the things that it must do:

- **Fetch instruction**: The CPU reads an instruction from memory.
- **Interpret instruction**: The instruction is decoded to determine what action is required.
- **Fetch data**: The execution of an instruction may require reading data from memory or an I/O module.
- **Process data**: The execution of an instruction may require performing some arithmetic or logical operation on data.
- **Write data**: The results of an execution may require writing data to memory or an I/O module.

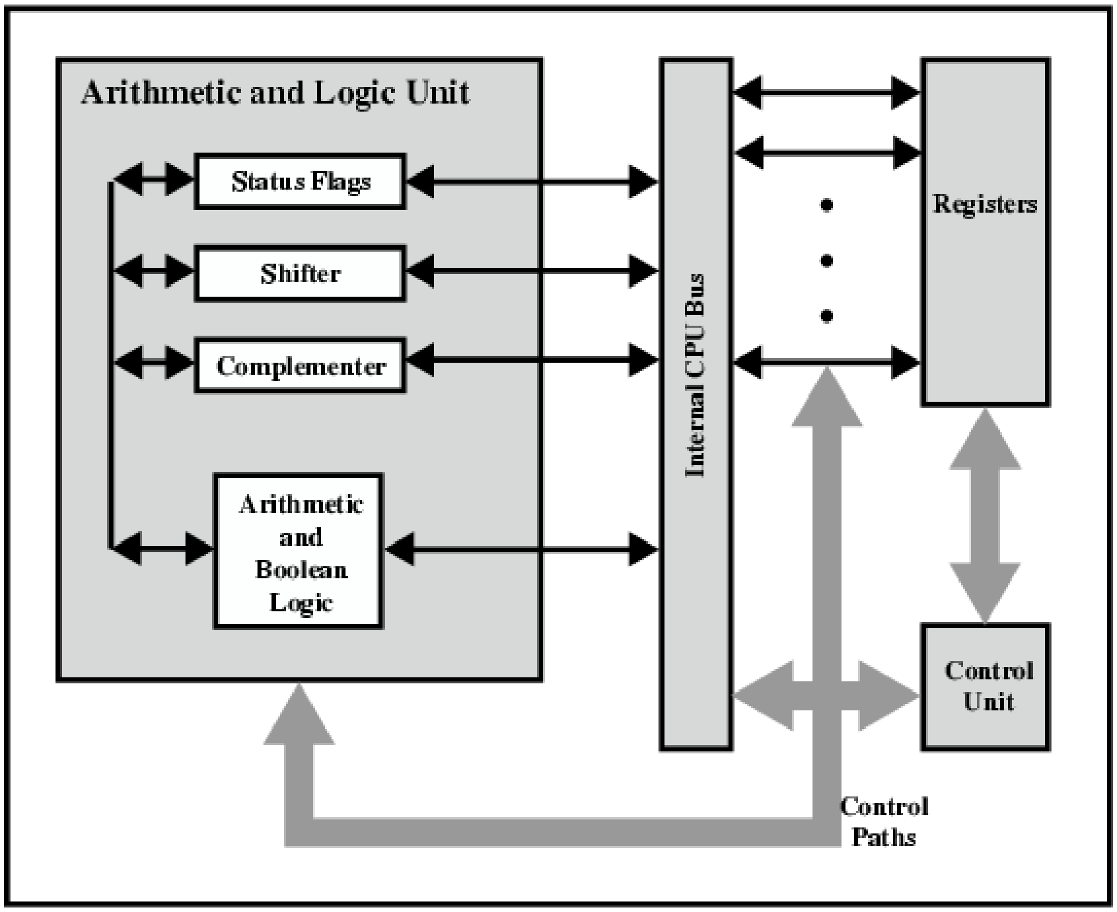
To do these things, it should be clear that the CPU needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the CPU needs a small internal memory.

[link] is a simplified view of a CPU, indicating its connection to the rest of the system via the system bus. You will recall (Lecture 1) that the major components of the CPU are an **arithmetic and logic unit** (ALU) and a **control unit** (CU). The ALU does the actual computation or processing of data. The control unit controls the movement of data and instructions into and out of the CPU and controls the operation of the ALU. In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called **registers**.



The CPU with the System Bus

[link] is a slightly more detailed view of the CPU. The data transfer and logic control paths are indicated, including an element labeled **internal CPU-bus**. This element is needed to transfer data between the various registers and the ALU because the ALU in fact operates only on data in the internal CPU memory.



CPU Internal Structure

Register organization

Within the CPU, there is a set of registers that function as a level of memory above main memory and cache in the hierarchy. The registers in the CPU perform two roles:

- **User-visible registers**: These enable the machine- or assembly-language programmer to minimize main memory references by optimizing use of registers.
- **Control and status registers**: These are used by the control unit to control the operation of the CPU and by privileged, operating system

programs to control the execution of programs.

There is not a clean separation of registers into these two categories. For example, on some machines the program counter is user visible (e.g., Pentium), but on many it is not (e.g., PowerPC). For purposes of the following discussion, however, we will use these categories.

User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the CPU executes. We can characterize these in the following categories:

- General purpose
- Data
- Address
- Condition codes

General-purpose registers: can be assigned to a variety of functions by the programmer. Sometimes their use within the instruction set is orthogonal to the operation. That is, any general--purpose register can contain the operand for any opcode. This provides true general-purpose register use. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point and stack operations. In some cases, general-purpose registers can be used for addressing functions (e.g.. register indirect, displacement). In other cases, there is a partial or clean separation between data registers and address registers.

Data registers may be used only to hold data and cannot be employed in the calculation of an operand address.

Address registers may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:

- **Segment pointers**: In a machine with segmented addressing, a segment register holds the address of the base of the segment. There may be multiple registers: for example, one for the operating system and one for the current process.
- **Index registers**: These are used for indexed addressing and may be autoindexed.
- **Stack pointer**: If there is user-visible stack addressing, then typically the stack is in memory and there is a dedicated register that points to the top of the slack. This allows implicit addressing; that is, push, pop, and other slack instructions need not contain an explicit stack operand.

Condition codes register (also referred to **as flags**): Condition codes are bits set by the CPU hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

Control and Status Registers

There are a variety of CPU registers that are employed to control the operation of the CPU. Most of these, on most machines, are not visible to the user. Some of them may be visible to machine instructions executed in a control or operating system mode.

Of course, different machines will have different register organizations and use different terminology. We list here a reasonably complete list of register types, with a brief description.

Four registers are essential to instruction execution:

- **Program counter (PC)**: Contains the address of an instruction to be fetched.
- **Instruction register (IR)**: Contains the instruction most recently fetched.

- **Memory address registers (MAR)**: Contains the address of a location in memory.
- **Memory buffer register (MBR)**: Contains a word of data lo be written to memory or the word most recently read.

Typically, the CPU updates the PC after each instruction fetch so that the PC always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC. The fetched instruction is loaded into an IR, where the opcode and operand specifiers are analyzed. Data are exchanged with memory using the MAR and MBR. In a bus-organized system, the MAR connects directly to the address bus, and the MBR connects directly to the data bus. User-visible registers, in turn, exchange data with the MBR.

The four registers just mentioned are used for the movement of data between the CPU and memory. Within the CPU, data must be presented to the ALU for processing. The ALU may have direct access to the MBR and user-visible registers. Alternatively, there may be additional buffering registers at the boundary to the ALU: these registers serve as input and output registers for the ALL and exchange data with the MBR and user-visible registers.

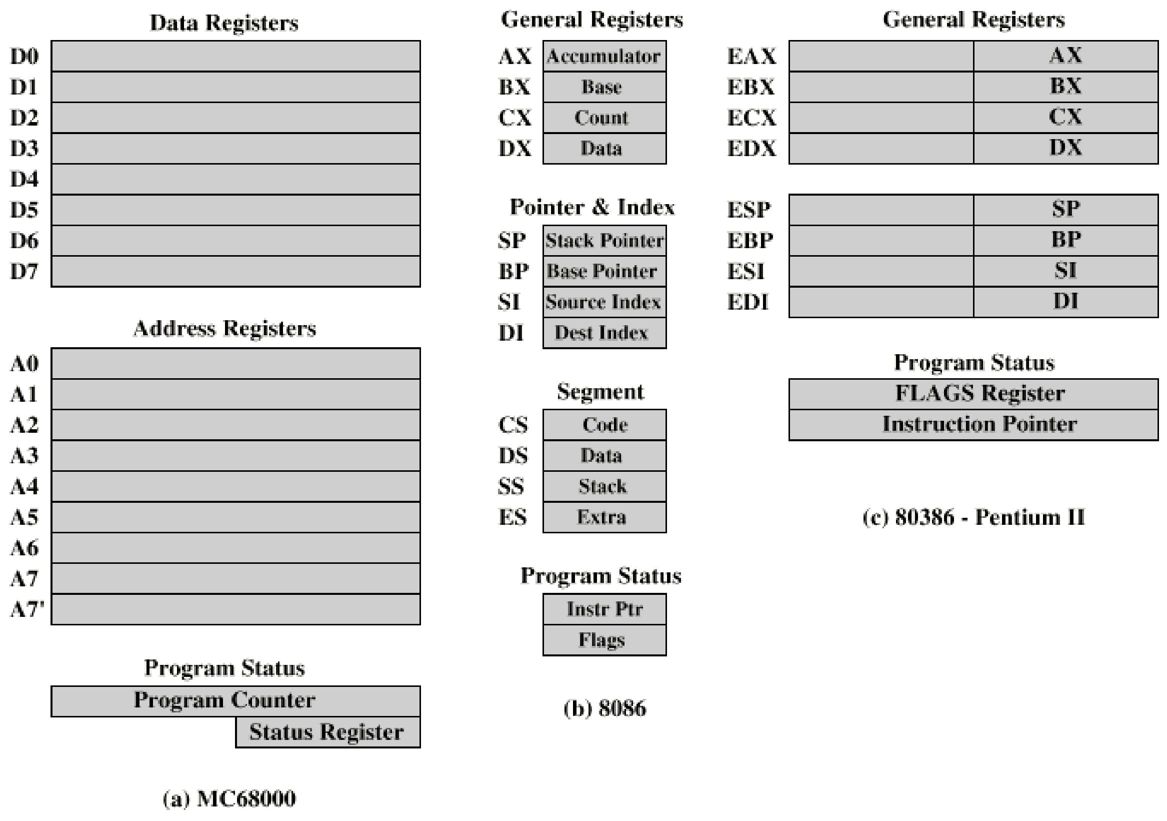
All CPU designs include a register or set of registers, often known as the **program status word** (PSW), that contain status information. The PSW typically contains condition codes plus other stains information. Common fields or flags include the following:

- **Sign**: Contains the sign bit of the result of the last arithmetic operation.
- **Zero**: Set when the result is 0.
- **Carry**: Set if an operation resulted in a carry (addition) into or borrow (sub-traction) out of a high-order hit. Used for multiword arithmetic operations.
- **Equal**: Set if a logical compare result is equality.
- **Overflow**: Used to indicate arithmetic overflow
- **Interrupt enable/disable**: Used to enable or disable interrupts.
- **Supervisor**: Indicates whether the CPU is executing in supervisor or user mode. Certain privileged instructions can be executed only in

supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

A number of other registers related to status and control might be found in a particular CPU design. In addition to the PSW, there may be a pointer to a block of memory containing additional status information (e.g., process control blocks).

Example Register Organizations:



Example of microprocessor registers organizations

Module 6: Control Unit Operation

1. Micro-Operation

The execution of a program consists of the sequential execution of instructions. Each instruction is executed during an instruction cycle made up of shorter sub-cycles (e.g., fetch, indirect, execute, interrupt). The performance of each sub-cycle involves one or more shorter operations, that is, micro-operations.

Micro-operations are the functional, or atomic, operations of a processor. In this section, we will examine micro-operations to gain an understanding of how the events of any instruction cycle can be described as a sequence of such micro-operations (Figure 6.1).

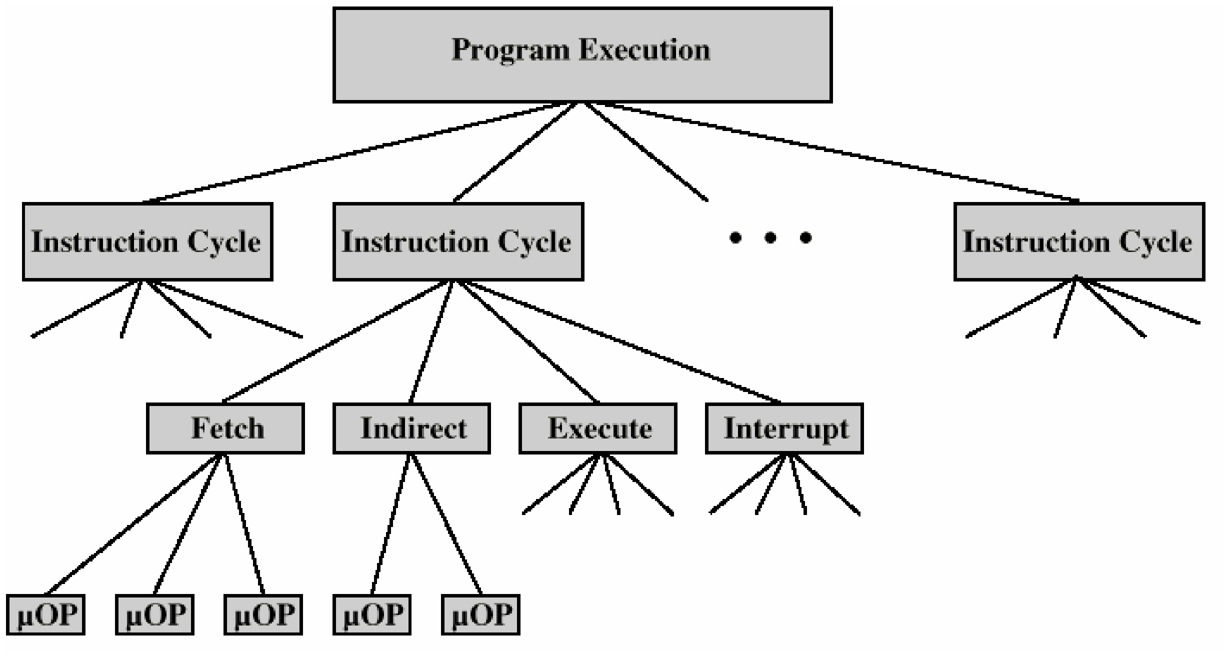


Figure 6.1 Constituent Elements of Program Execution

1.1 The Fetch Cycle

We begin by looking at the fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. Four registers are involved:

- Memory address register (MAR): Is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- Memory buffer register (MBR): Is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.
- Program counter (PC): Holds the address of the next instruction to be fetched.
- Instruction register (IR): Holds the last instruction fetched.

Let us look at the sequence of events for the fetch cycle from the point of view of its effect on the processor registers. An example appears in Figure 6.2.

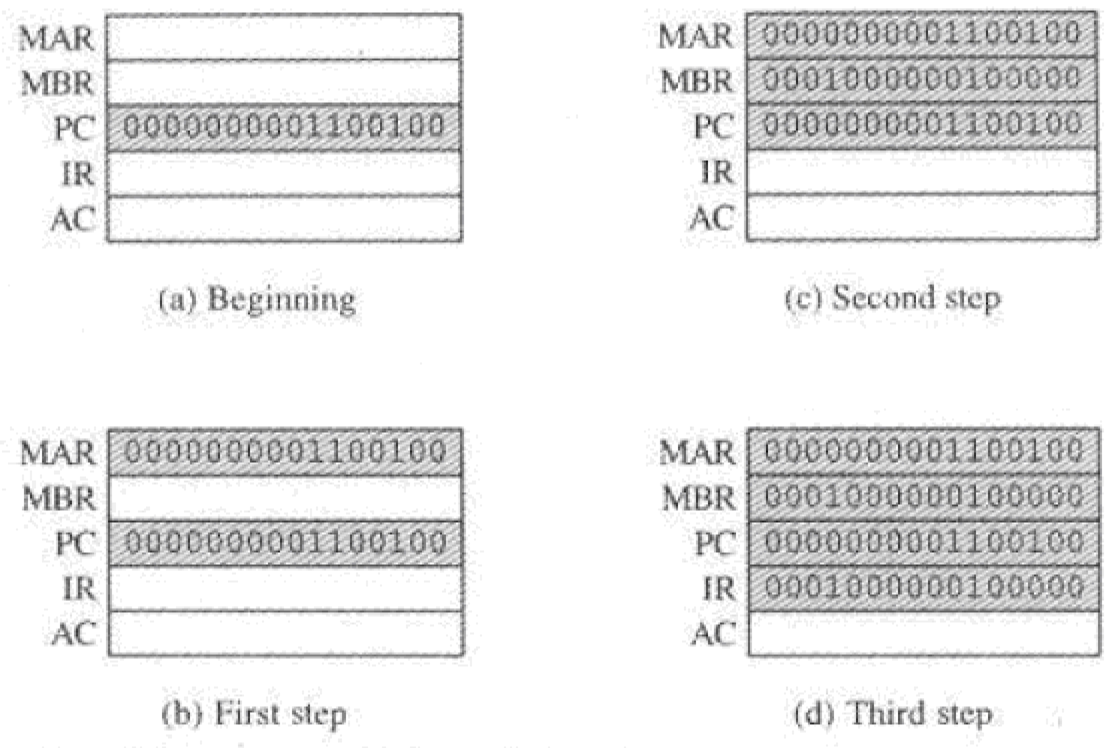


Figure 6.2 Sequence of Events, Fetch Cycle

- At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter (PC); in this case, the address is 1100100.
- The first step is to move that address to the memory address register (MAR) because this is the only register connected lo the address lines of the system bus.
- The second step is to bring in the instruction. The desired address (in the MAR) is placed on the address bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR). We also need to increment the PC by 1 to get ready for the next instruction. Because these two actions (read word from memory, add 1 to PC) do not interfere with each other, we can do them simultaneously to save time.
- The third step is to move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.

Thus, the simple fetch cycle actually consists of three steps and four micro-operations. Each micro-operation involves the movement of data into or out of a register. So long as these movements do not interfere with one another, several of them can take place during one step, saving lime. Symbolically, we can write this sequence of events as follows:

t1: MAR <= (PC)

t2: MBR <= Memory

PC <= (PC) + l

t3: IR <= (MBR)

where l is the instruction length. We need to make several comments about this sequence. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation (t1, t2, t3) represents successive time units. In words, we have

- First time unit: Move contents of PC to MAR.
- Second time unit:
    - Move contents of memory location specified by MAR to MBR.
    - Increment by l the contents of the PC.
- Third time unit: Move contents of MBR to IR.

Note that the second and third micro-operations both take place during the second time unit. The third micro-operation could have been grouped with the fourth without affecting the fetch operation:

t1: MAR <= (PC)

t2: MBR <= Memory

t3: PC <= (PC) + l

IR <= (MBR)

The groupings of micro-operations must follow two simple rules:

1. The proper sequence of events must be followed. Thus (MAR <= (PC)) must precede (MBR <= Memory) because the memory read operation makes use of the address in the MAR.

2. Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable. For example, the micro-operations (MBR <= Memory) and (IR <= MBR) should not occur during the same time unit.

A final point worth noting is that one of the micro-operations involves an addition. To avoid duplication of circuitry, this addition could be performed by the ALU. The use of the ALU may involve additional micro-operations, depending on the functionality of the ALU and the organization of the processor.

1.2 The Indirect Cycle

Once an instruction is fetched, the next step is to fetch source operands. Continuing our simple example, let us assume a one-address instruction format, with direct and indirect addressing allowed. If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle. The data flow includes the following micro-operations:

t1: MAR <= (IR (Address))

t2: MBR <= Memory

t3: IR(Address) <= (MBR(Address))

The address field of the instruction is transferred to the MAR. This is then used to fetch the address of the operand. Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.

The IR is now in the same state as if indirect addressing had not been used, and it is ready for the execute cycle. We skip that cycle for a moment, to consider the interrupt cycle.

1.3 The Interrupt Cycle

At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs. The nature of this cycle varies greatly from one machine to another. We present a very simple sequence of events, we have

t1 : MBR <= (PC)

t2 : MAR <= Save_Address

PC <= Routine_Address

t3: Memory <= (MBR)

In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing routine. These two actions may each be a single micro-operation. However, because most processors provide multiple types and/or levels of interrupts, it may lake one or more additional micro-operations to obtain the save_address and the routine_address before they can be transferred to the MAR and PC, respectively. In any case, once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

1.4 The Execute Cycle

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and, in each ease, the same micro-operations are repealed each time around. This is not true of the execute cycle. For a machine with N different opcodes, there are N different sequences of micro-operations that can occur. Let us consider several hypothetical examples.

First, consider an add instruction:

ADD R1, X

which adds the contents of the location X to register Rl. The following sequence of micro-operations might occur:

t1: MAR <= (IR(address))

t2: MBR <= Memory

t3: Rl <= (Rl) + (MBR)

We begin with the IR containing the ADD instruction. In the first step, the address portion of the IR is loaded into the MAR. Then the referenced memory location is read. Finally, the contents of R1 and MBR are added by the ALU. Again, this is a simplified example. Additional micro-operations may be required to extract the register reference from the IR and perhaps to stage the ALU inputs or outputs in some intermediate registers.Let us look at two more complex examples. A common instruction is increment and skip if zero:

ISZ X

The content of location X is incremented by 1. If the result is 0, the next instruction is skipped. A possible sequence of micro-operations is

t1: MAR <= (CR(address))

t2: MBR <= Memory

t3: MBR <= (MBR) - 1

t4: Memory <= (MBR)

If ((MBR) = 0) then (PC <= (PC) + I)

The new feature introduced here is the conditional action. The PC is incremented if (MBR) = 0; this test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory.

Finally, consider a subroutine call instruction. As an example, consider a branch-and-save-address instruction:

BSA X

The address of the instruction that follows the BSA instruction is saved in location X, and execution continues al location X - l. The saved address will later be used for return. This is a straightforward technique for providing subroutine calls. the following micro-operations suffice:

t1 : MAR <= (IR(address))

MBR <= (PC)

t2: PC <= (IR(address)) Memory <= (MBR)

t3: PC <= (PC) + I

The address in the PC at the start of the instruction is the address of the next instruction in sequence. This is saved at the address designated in Ihe IK. The latter address is also incremented to provide the address of the instruction for the next instruction cycle.

1.5 The Instruction Cycle

We have seen that each phase of the instruction cycle can be decomposed into a sequence of elementary micro-operations. In our example, there is one sequence each for the fetch, indirect, and interrupt cycles, and, for the execute cycle, there is one sequence of micro-operations for each opcode. To complete the picture, we need to tie sequences of micro-operations together, and this is done in Figure 6.3.

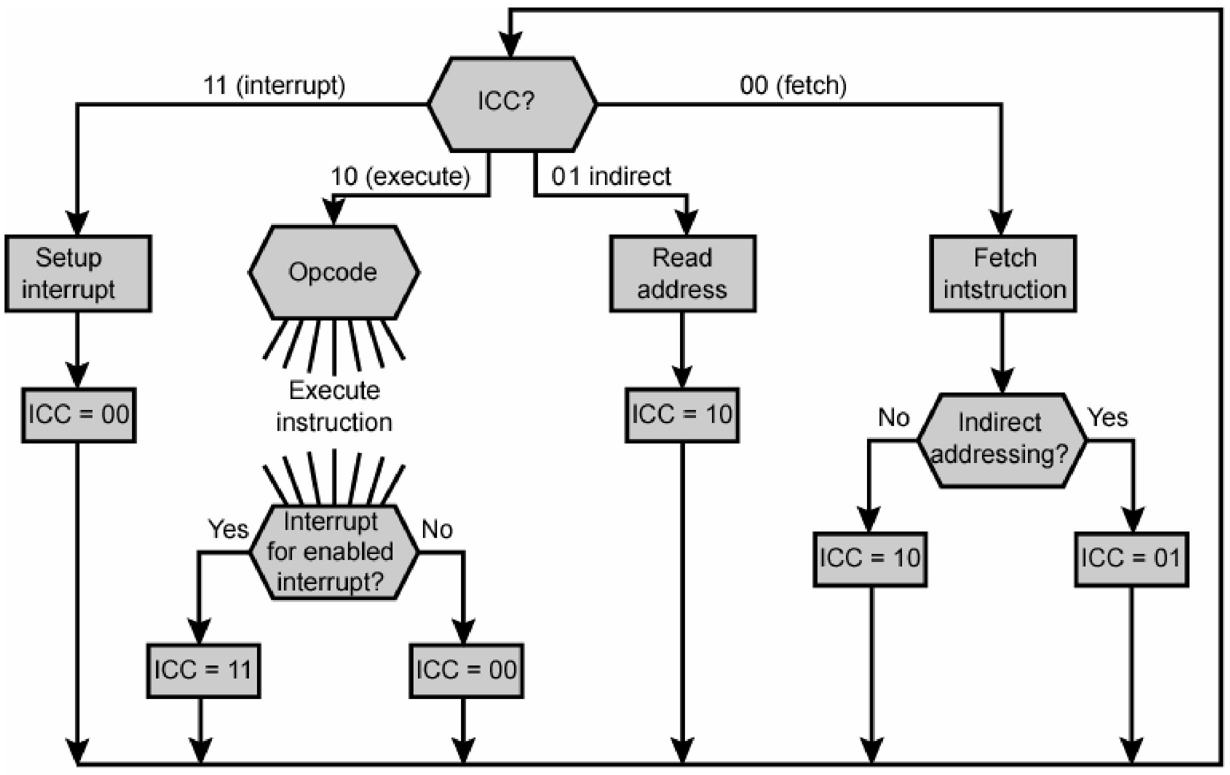


Figure 6.3 Flowchart for Instruction Cycle

We assume a new 2-bit register called the instruction cycle code (ICC). The ICC designates the state of the processor in terms of which portion of the cycle it is in:

00: Fetch

01: Indirect

10: Execute

11: Interrupt

At the end of each of the four cycles, the ICC is set appropriately. The indirect cycle is always followed by the execute cycle. The interrupt cycle is always followed by the fetch cycle. For both the execute and fetch cycles, the next cycle depends on the state of the system.

Thus, the flowchart of Figure 6.3 defines the complete sequence of micro-operations, depending only on the instruction sequence and the interrupt pattern. Of course, this is a simplified example. The flowchart for an actual processor would be more complex. In any case, we have reached the point in our discussion in which the operation of the processor is defined as the performance of a sequence of micro-operations. We can now consider how the control unit causes this sequence to occur.

2. Control Of The Processor

2.1 Functional Requirements

As a result of our analysis in the preceding section, we have decomposed the behavior or functioning of the processor into elementary operations, called micro-operations. By reducing the operation of the processor to its most fundamental level, we are able to define exactly what it is that the control unit must cause to happen. Thus, we can define the functional requirements for the control unit those functions that the control unit must perform. A definition of these functional requirements is the basis for the design and implementation of the control unit.

With the information at hand, the following three-step process leads to a characterization of the control unit:

1. Define the basic elements of the processor.
2. Describe the micro-operations that the processor performs.
3. Determine the functions that the control unit must perform to cause the micro-operations to be performed.

We have already performed steps 1 and 2. Let us summarize the results. First, the basic functional elements of the processor are the following:

- ALU
- Registers
- Internal data paths
- External data paths
- Control unit

Some thought should convince you that this is a complete list. The ALU is the functional essence of the computer. Registers are used to stoic data internal to the processor. Some registers contain status information needed to manage instruction sequencing (e.g., a program status word). Others contain data that go to or come from the ALU, memory, and I/O modules. Internal data paths are used to move data between registers and between register and ALU. External data paths link registers to memory and I/O modules, often by means of a system bus. The control unit causes operations to happen within the processor.

The execution of a program consists of operations involving these processor elements. As we have seen, these operations consist of a sequence of micro-operations. All micro-operations fall into one of the following categories:

- Transfer data from one register to another.
- Transfer data from a register to an external interface (e.g., system bus).
- Transfer data from an external interface lo a register.
- Perform an arithmetic or logic operation, using registers for input and output.

All of the micro-operations needed to perform one instruction cycle, including all of the micro-operations to execute every instruction in the instruction set, fall into one of these categories.

We can now be somewhat more explicit about the way in which the control unit functions. The control unit performs two basic tasks:

- Sequencing: The control unit causes the processor lo step through a series of micro-operations in the proper sequence, based on the program being executed.
- Execution: The control unit causes each micro-operation to be performed.

The preceding is a functional description of what the control unit does. The key to how the control unit operates is the use of control signals.

2.2 Control Signals

We have defined the elements that make up the processor (ALU, registers, data paths) and the micro-operations that are performed. For the control unit to perform its function, it must have inputs that allow it to determine the slate of the system and outputs that allow it to control the behavior of the system. These are the external specifications of the control unit. Internally, the control unit must have the logic required lo perform its sequencing and execution functions.

Figure 6.4 is a general model of the control unit, showing all of its inputs and outputs.

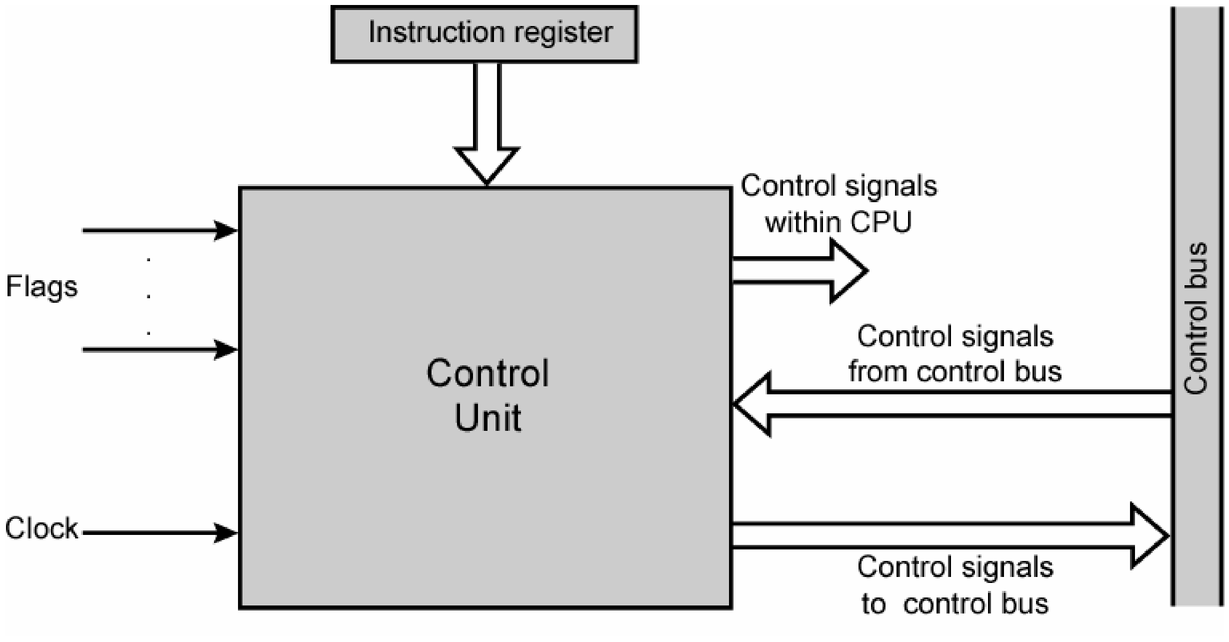

Instruction register
Flags
Clock
Control Unit
Control signals within CPU
Control signals from control bus
Control signals to control bus
Control bus

Figure 6.4 Model of Control Unit

The inputs are as follows:

- Clock: This is how the control unit "keeps time." The control unit causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse. This is sometimes referred to as the processor cycle time. or the clock cycle lime.

- Instruction register: The opcode of the current instruction is used lo determine which micro-operations lo perform during the execute cycle.
- Flags: These are needed by the control unit to determine the status of the processor and the outcome of previous ALU operations. For example, for the increment-and-skip-if-zero (ISZ) instruction, the control until will increment the PC if the zero flag is set.
- Control signals from control bus: The control bus portion of the system bus pro-vides signals to the control unit, such as interrupt signals and acknowledgments.

The outputs are as follows:

- Control signals within the processor: These are two types: those that cause data to be moved from one register to another, and those that activate specific ALU functions.
- Control signals to control bus: These are also of two types: control signals lo memory, and control signals lo the I/O modules.

The new element that has been introduced in this figure is the control signal. Three types of control signals are used: those that activate an ALU function, those that activate a data path, and those that are signals on the external system bus or other external interface. All of these signals are ultimately applied directly as binary inputs lo individual logic gates.

Let us consider again the fetch cycle to see how the control unit maintains control. The control unit keeps track of where it is in the instruction cycle. At a given point, it knows that the fetch cycle is to be performed next. The first step is to transfer the contents of the PC to the MAR. The control unit does this by activating the control signal that opens the gates between the bits of the PC and the bits of the MAR. The next step is to read a word from memory into the MBR and increment the PC. The control unit does this by sending the following control signals simultaneously:

- A control signal that opens gates, allowing the contents of the MAR onto the address bus
- A memory read control signal on the control bus
- A control signal that opens the gates, allowing the contents of the data bus to be stored in the MBR

- Control signals to logic that add 1 to the contents of the PC and store the result back to the PC

Following this, the control unit sends a control signal that opens gates between the MBR and the IR.

This completes the fetch cycle except for one thing: The control unit must decide whether to perform an indirect cycle or an execute cycle next. To decide this, it examines the IR to see if an indirect memory reference is made.

The indirect and interrupt cycles work similarly. For the execute cycle, the control unit begins by examining the opcode and. on the basis of that, decides which sequence of micro-operations to perform for the execute cycle.

A Control Signals Example

To illustrate the functioning of the control unit, let us examine a simple example. Figure 6.5 illustrates the example.

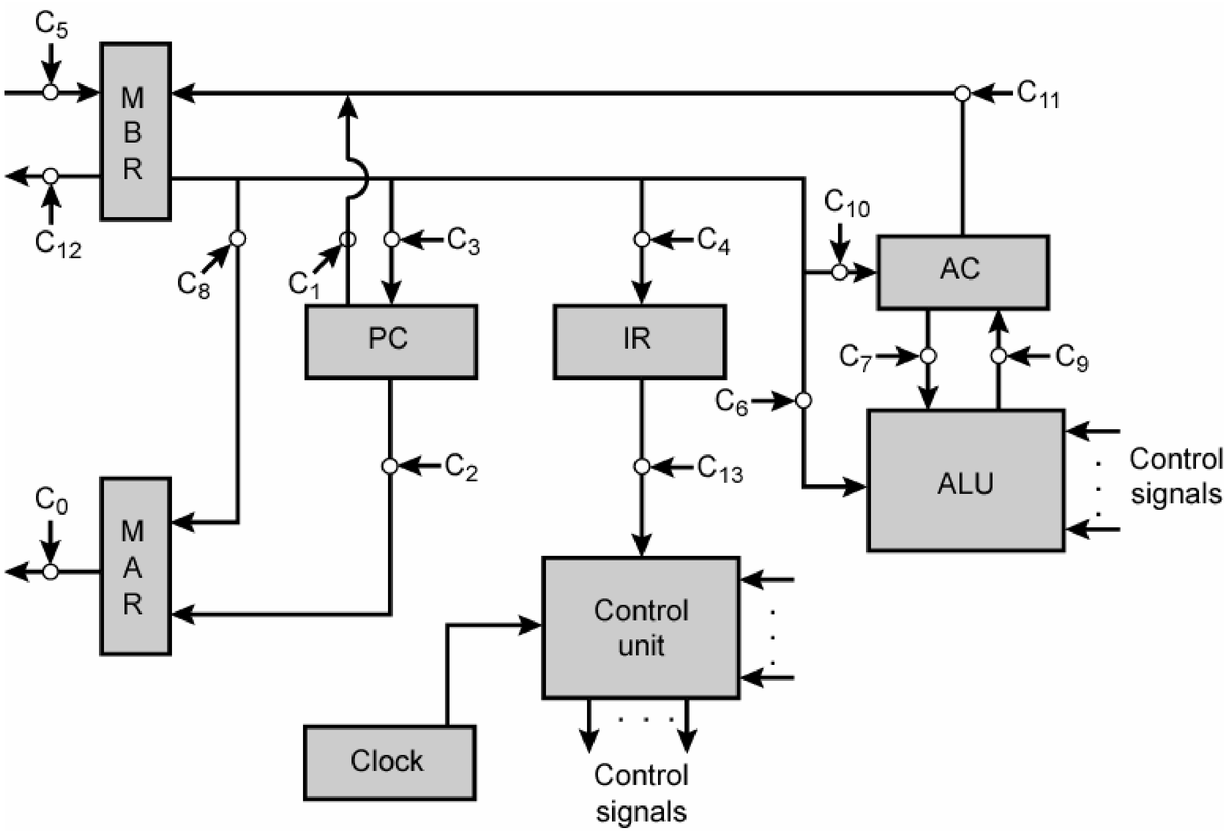


Figure 6.5 Data Paths and Control Signals

This is a simple processor with a single accumulator. The data paths between elements are indicated. The control paths for signals emanating from the control unit are not shown, but the terminations of control signals are labeled Ci and indicated by a circle. The control unit receives inputs from the clock, the instruction register, and flags. With each clock cycle, the control unit reads all of its inputs and emits a set of control signals. Control signals go to three separate destinations:

- Data paths: The control unit controls the internal How of data. For example, on instruction fetch, the contents of the memory buffer register are transferred to the instruction register. For each path to be controlled, there is a gate (indicated by a circle in the figure). A control signal from the control unit temporarily opens the gate to let data pass.
- ALU: The control unit controls the operation of the ALU by a set of control signals. These signals activate various logic devices and gates within the ALU.

- System bus: The control unit sends control signals out onto the control lines of the system bus (e.g., memory READ).

The control unit must maintain knowledge of where it is in the instruction cycle. Using this knowledge, and by reading all of its inputs, the control unit emits a sequence of control signals that causes micro-operations to occur.

2.3 Internal Processor Organization

Figure 6.5 indicates the use of a variety of data paths. The complexity of this type of organization should be clear. More typically, some sort of internal bus arrangement. Using an internal processor bus, Figure 6.5 can be rearranged as shown in Figure 6.6.

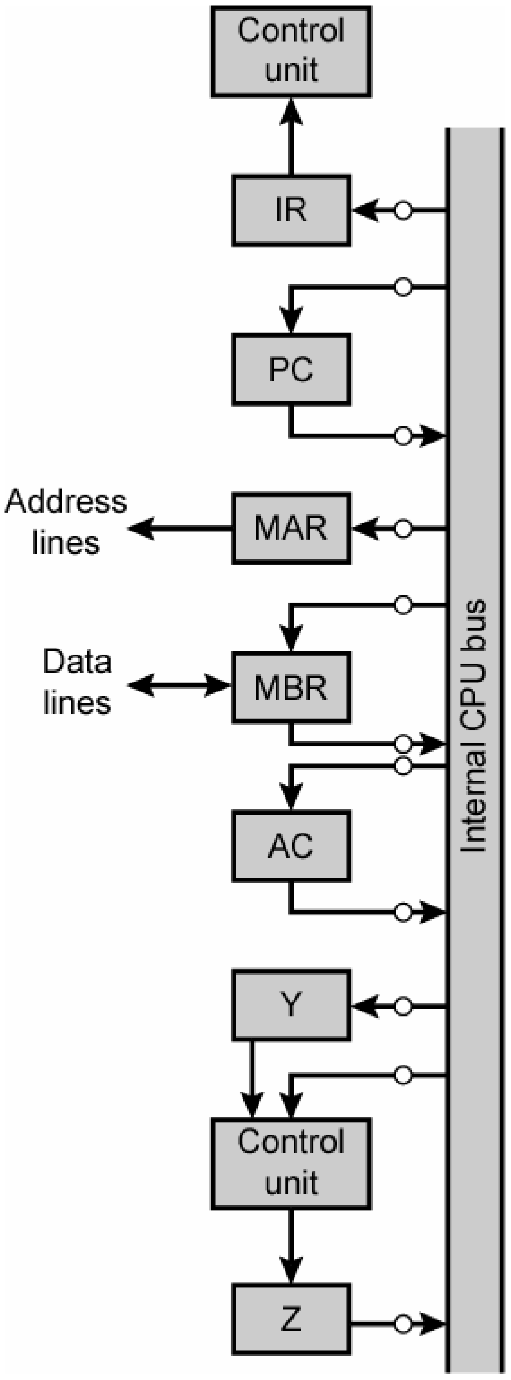


Figure 6.6 CPU with Internal Bus

A single internal bus connects the ALU and all processor registers. Gates and control signals are provided for movement of data onto and off the bus from each register. Additional control signals control data transfer to and from the system (external) bus and the operation of the ALU.

Two new registers, labeled Y and Z have been added to the organization. These are needed for the proper operation of the ALU. When an operation involving two operands is performed, one can be obtained from the internal bus, but the other must be obtained from another source. The AC could be used for this purpose, but this limits the flexibility of the system and would not work with a processor with multiple general-purpose registers. Register Y provides temporary storage for the other input. The ALU is a combinatorial circuit with no internal storage, Thus, when control signals activate an ALU function, the input to the ALU is transformed to the output. Thus, the output of the ALU cannot be directly connected to the bus, because this output would feed back to the input. Register Z provides temporary output storage, with this arrangement, an operation to add a value from memory to the AC would have the following steps:

t1: MAR <= (IR(address))

t2: MBR <= Memory

t3: Y <= (MBR)

t4: Z <= (AC0 + (Y)

t5: ac <= (z)

Other organizations are possible, but, in general, some sort of internal bus or set of internal buses is used. The use of common data paths simplifies the interconnection layout and the control of the processor. Another practical reason for the use of an internal bus is to save space. Especially for microprocessors, which may occupy only a 1/4-inch square piece of silicon, space occupied by internal connections must be minimized.

Module 7: Microprogramming

1. Basis Concepts

- Micro-operations: We have already seen that the programs are executed as a sequence of instructions, each instruction consists of a series of steps that make up the instruction cycle fetch, decode, etc. Each of these steps are, in turn, made up of a smaller series of steps called micro-operations.

- Micro-operation execution: Each step of the instruction cycle can be decomposed into micro-operation primitives that are performed in a precise time sequence. Each micro-operation is initiated and controlled based on the use of control signals / lines coming from the control unit.

- Controller the data to move from one register to another

- Controller the activate specific ALU functions

- Micro-instruction: Each instruction of the processor is translated into a sequence of lower-level micro-instructions. The process of translation and execution are to as microprogramming
- Microprogramming: The concept of microprogramming was developed by Maurice Wilkes in 1951, using diode matrices for the memory element. A microprogram consist of a sequence of micro-instructions in a microprogramming.
- Microprogrammed Control Unit is a relatively logic circuit that is capable of sequencing through micro-instructions and generating control signal to execute each micro-instruction.
- Control Unit: The control Unit is an important portion of the processor.

The control unit issues control signals external to the processor to cause data echange with memory and I/O unit. The control Unit issues also control signals internal to the processor to move data between registres, to perform the ALU and other internal operations in processor. In a hardwired control unit, the control signals are generated by a micro-instruction are used to controller register transfers and ALU operations. Control Unit design is then the collection and the implementation of all of the needed control signals for the micro-instruction executions.

2. Control unit design approaches

How can we use the concept of microprogramming to implement a Control Unit ? There are two approaches of Control Unit Design and implementation:

- Microprogrammed implementation

- Hardwired logic implementation

The figure 7.2 illustrated the control unit inputs. Two techniques have been used to implemente the CU. In a hardwired implementation, the control unit is essentially considered as a logic circuit. Its input logic signals are transformed into the set of ouput logic signals, which are the control signals. The approach of microprogrammed implementation is studied in this section.

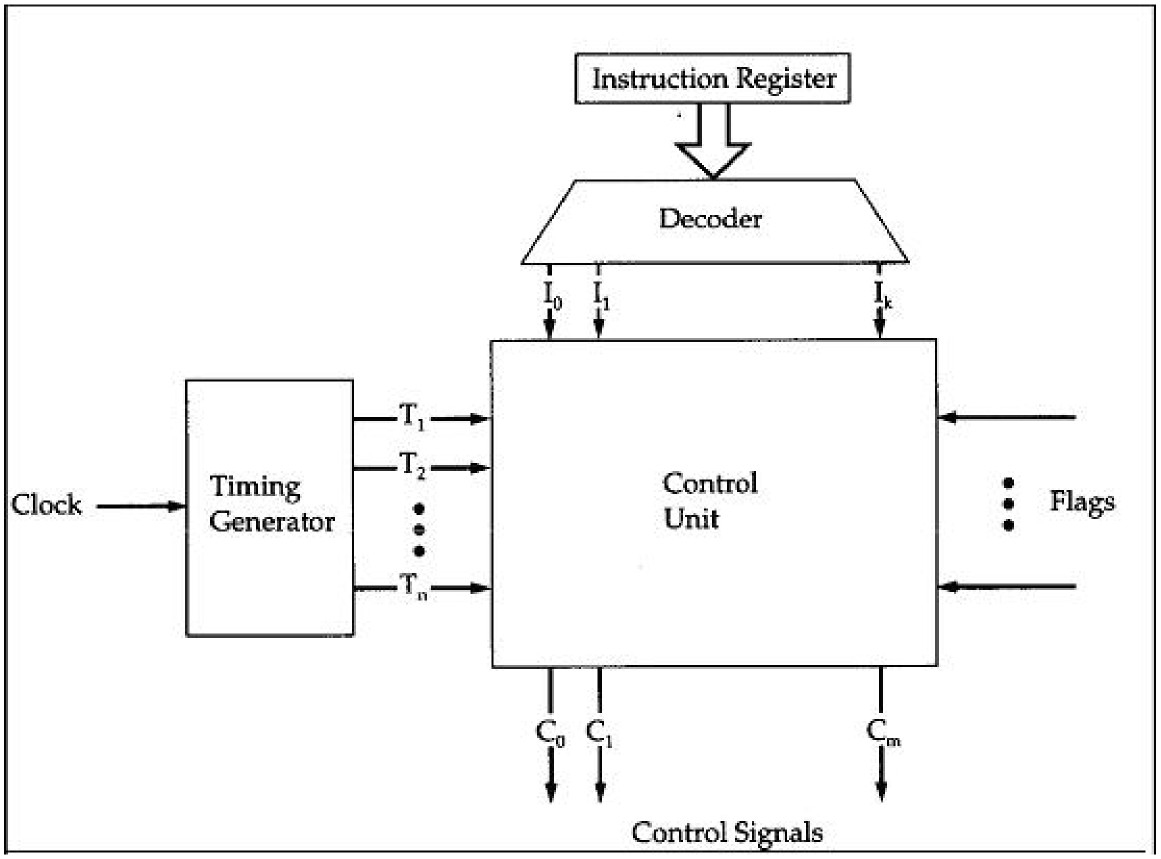


Figure 7.2 Control unit with decoded inputs

2.1 Approach of microprogrammed control unit

Principe:

- The control signal values for each microoperation are stored in a memory.

- Reading the contents of the control store in a prescribed order is equivalent to sequencing through the microoperations

- Since the “microprogram” of microoperations and their control signal values are stored in memory, this is a microprogrammed unit.

Remarks:

- Are more systematic with a well defined format?

- Can be easily modified during the design process?
- Require more components to implement
- Tend to be slower than hardwired units (due to having to perform memory read operations)

2.2 Approach of hardwired logic

Principe:

- The Control Unit is viewed and designed as a combinatorial and sequential logic circuit.

- The Control Unit is implemented by using any of a variety of “standard” digital logic techniques. The logic circuit generate the fixed sequences of control signals

- This approach is used to generate fixed sequences of control signals with the higher speed.

Remarks:

- The principle advantages are a high(er) speed operation and the smaller implementations (component counts)
- The modifications to the design can be hard to do
- This approach is favored in RISC style designs

3. Microprogrammed Control Unit

The ideal of microprogrammed Control Unit is that the Control Unit design must include the logics for sequencing through micro-operations, for executing micro-operation, for executing micro-instructions, for interpreting opcodes and for making decision based on ALU flags. So the design is relatively inflexible. It is difficul to change the design if one wishes to add a new machine instruction.

The principal disadvantage of a microprogrammed control unit is that it will be slower than hardwired unit of comparable technology. Despite this, microprogramming is the dominant technique for implementing control unit in the contemporary CISC processor, due to its ease of implementation.

The control unit operates by performing consecutive control storage reads to generate the next set of control function outputs. Performing the series of control memory accesses is, in effect, executing a program for each instruction in the machine's instruction set -- hence the term microprogramming.

The two basic tasks performed by a microprogrammed control unit are as follows:

- Micro-instruction sequencing: the microprogrammed control unit get the next mico-instruction from the control memory

- Micro-instruction execution: the microprogrammed control unit generate the control signals needed to execute the micro-instruction.

The control unit design must consider both affect the format of the micro-instruction and the timing of the control unit.

3.1 Micro-instruction Sequencing

Two problems are involved in the design of a micro-instruction sequencing technique is the size of micro-instruction and the address-generation time. The first concern is obvious minimizing the size of the control memory. The second concern is simply a desire to execute microinstruction as fast as possible.

In executing a microprogram, the address of the next microinstruction to be executed is one of these categories:

- Determined by instruction register

- Next sequential address

- Branch.

It is important to design compact time-efficient techniques for micro-instruction branching.

- Sequencing technique

Three general categories for a control memory address are as follows:

- Two address fields

- Single address field

- Variable format

In Figure 7.3, the branch control logic with a single address field is illustrated.

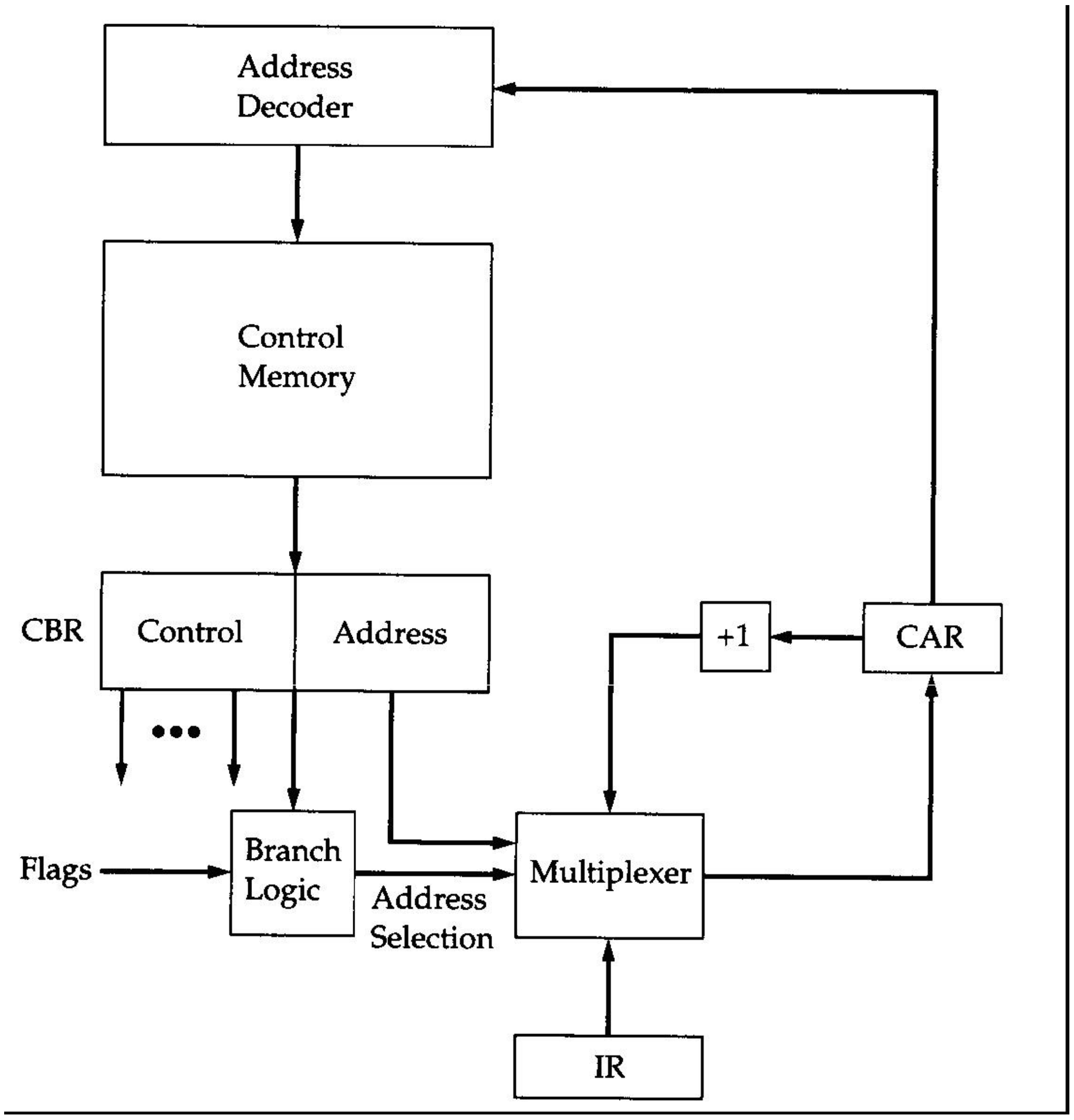
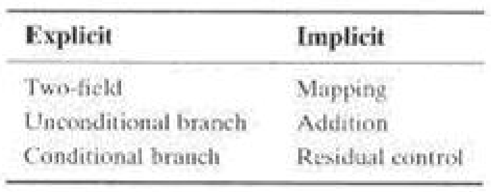


Figure 7.3. Branch Control unit of Microprogrammed Control Unit with with a single address field

- Address generation

The problem is to consider the various ways in which the next address can be derived or computed. The various techniques of the address generation is geven in the following.



Explicit	Implicit
Two-field	Mapping
Unconditional branch	Addition
Conditional branch	Residual control

Table 1: Microinstruction Address Generation techiques

3.2 Micro-instruction Execution

The microinstruction cycle is the basic event on a microprogrammed processor. Each cycle is made up the two parts: fetch and execute. This section deals with the execution of microinstruction. The effect of the execution of a microinstruction is to generate control signals for both the internal control to processor and the external control to processor.

A organization of a control unit is shown in Figure 7.4

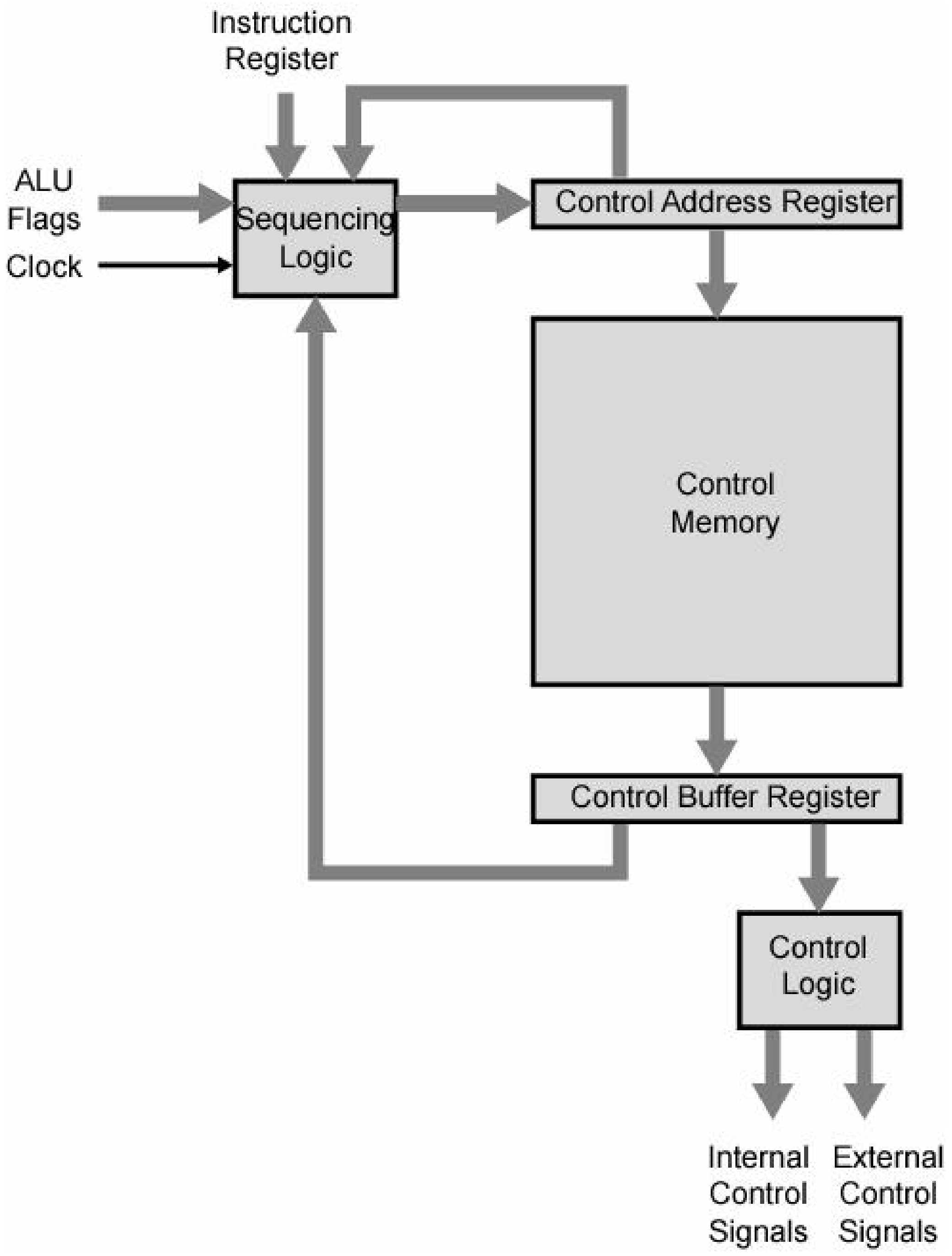


Figure 7.4. Microprogrammed Control Unit Organization

4. Classification of Micro-instructions

Microinstruction can be classified in variety of ways in which the designer must choose the parallel "power" of each instruction. There are the following.

– Vertical microprogramming: Each microinstruction specifies a single (or few) microoperations to be performed

– Horizontal microprogramming: Each microinstruction specifies many different

microoperations to be performed in parallel.

- Vertical microprogramming

– Width is narrow: n control signals can be encoded into log2n control bits

– Limited ability to express parallelism

– Considerable encoding of control information requires external memory word decoder to identify the exact control line being manipulated

- Horizontal microprogramming

– Wide memory word

– High degree of parallel operations are possible

– Little to no encoding of control information

- Compromise technique

– Divide control signals into disjoint groups

– Implement each group as a separate field in the memory word

– Supports reasonable levels of parallelism without too much complexity

- Second compromise: nanoprogramming

– Use a 2-level control storage organization

– Top level is a vertical format memory

Output of the top level memory drives the address register of the bottom (nano-level) memory

– Nanomemory uses the horizontal formal. The produces the actual control signal outputs

– The advantage to this approach is significant saving in control memory size (bits)

– Disadvantage is more complexity and slower operation (doing 2 memory accesses fro each microinstruction).

- Microprogramming applications

- For the typically large microprocessor systems today:

+ There are many instructions and associated register level hardware

+ There are many control point to be manipulated.

- Emulation

– The use of a microprogram on one machine to execute programs originally written to run on another machine.

– By changing the microcode of a machine, you can make it execute software from another machine.

Module 8: Instruction Pipelining

1. Pipelining

1.1 Basic concepts

An instruction has a number of stages. The various stages can be worked on simultanously through various blocks of production. This is a pipeline. This process is also referred as instruction pipeling. Figure 8.1 shown the pipeline of two independent stages: fetch instruction and execusion instruction. The first stage fetches an instruction and buffers it. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This process will speed up instruction execution

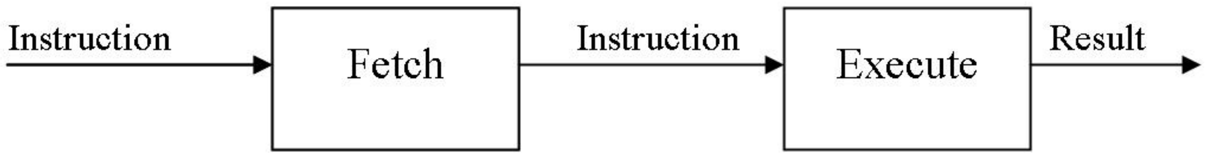


Figure 8.1. Two stages Instruction Pipeline

1.2 Pipeline principle

The decomposition of the instruction processing by 6 stages is the following.

- Fetch Instruction (FI): Read the next expected introduction into a buffer

- Decode Instruction (DI): Determine the opcode and the operand specifiers

- Calculate Operands (CO): Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect or other forms of address calculations.

- Fetch Operands (FO): Fetch each operand from memory. Operands in register need not be fetched.

- Execute Instruction (EI): Perform the indicated operation and store the result, if any, in the specified destination operand location.

- Write Operand (WO): Store result in memory.

Using the assumption of the equal duration for various stages, the figure 8.2 shown that a six stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

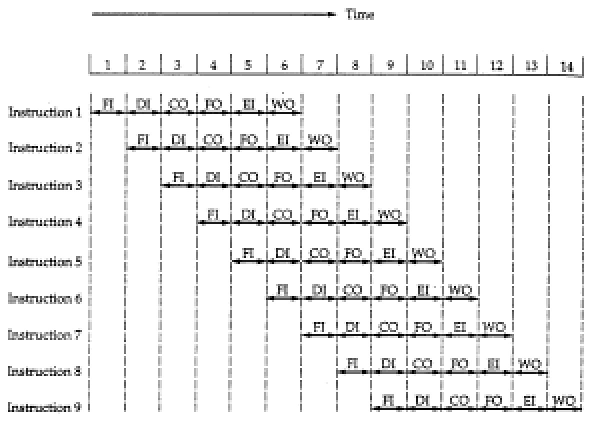


Figure 8.2. Timing diagram for instruction pipeline operation.

Also the diagram assumes that all of the stages can be performed in parallel, in particular, it is assumed that there are no memory conflicts. The processor make use of instruction pipelining to speed up executions, pipeling invokes breaking up the instruction cycle into a number of separate stages in a sequence. However the occurrence of branches and independencies between instruction complates the design and use of pipeline.

2. Pipeline Performance and Limitations

With the pipeling approach, as a form of parallelism, a “good” design goal of any system is to have all of its components performing useful work all of the time, we can obtain a high efficiency. The instruction cycle state diagram clearly shows the sequence of operations that take place in order to execute a single instruction.

This strategy can give the following:

- Perform all tasks concurrently, but on different sequential instructions

– The result is temporal parallelism.

– Result is the instruction pipeline.

2.1 Pipeline performance

In this subsection, we can show some measures of pipeline performance based on the book “Computer Organization and Architecture: Designing for Performance”, 6th Edition by William Stalling.

The cycle time   of an instruction pipeline can be determined as:

$T = \max[T_i] + d = T_m + d$ with 1 i k

where:

T_m = Maximun stage delay through stage

k = number of stages in instruction pipeline

d = time delay of a latch.

In general, the time delay d is equivalent to a clock pulse and T_m >> d. Suppose that n instruction are processed with no branched.

- The total time required T_k to execute all n instruction is:

T_k= [k + (n-1)]

- The speedup factor for the instruction pipeline compared to execution without the pipeline is defined as:

$$S_K = \frac{T_1}{T_K} = \frac{\text{nk}\tau}{[k+(n-1)]\tau} = \frac{\text{nk}}{k+(n-1)}$$

- An ideal pipeline divides a task into k independent sequential subtasks

– Each subtask requires 1 time unit to complete

– The task itself then requires k time units tocomplete. For n iterations of the task, the execution times will be:

– With no pipelining: nk time units

– With pipelining: k + (n-1) time units

Speedup of a k-stage pipeline is thus

S = nk / [k+(n-1)] ==> k (for large n)

2.2 Pipeline Limitations

Several factors serve to limit the pipeline performance. If the six stage are not of equal duration, there will be some waiting involved at various pipeline stage. Another difficulty is the condition branch instruction or the unpredictable event is an interrupt. Other problem arise that the memory conflicts could occur. So the system must contain logic to account for the type of conflict.

- Pipeline depth

- Data dependencies also factor into the effective length of pipelines

- Logic to handle memory and register use and to control the overall pipeline increases significantly with increasing pipeline depth

– If the speedup is based on the number of stages, why not build lots of stages?

– Each stage uses latches at its input (output) to buffer the next set of inputs

+ If the stage granularity is reduced too much, the latches and their control become a significant hardware overhead

+ Also suffer a time overhead in the propagation time through the latches

- Limits the rate at which data can be clocked through the pipeline

- Data dependencies

– Pipelining must insure that computed results are the same as if computation was performed in strict sequential order

– With multiple stages, two instructions “in execution” in the pipeline may have data dependencies. So we must design the pipeline to prevent this.

– Data dependency examples:

A = B + C

D = E + A

C = G x H

A = D / H

Data dependencies limit when an instruction can be input to the pipeline.

- Branching

One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to initial stages of the pipeline. However, 15-

20% of instructions in an assembly-level stream are (conditional) branches. Of these, 60-70% take the branch to a target address. Until the instruction is actually executed, it is impossible to determin whether the branch will be taken or not.

- Impact of the branch is that pipeline never really operates at its full capacity.

– The average time to complete a pipelined instruction becomes

Tave =(1-pb)1 + pb[pt(1+b) + (1-pt)1]

– A number of techniques can be used to minimize the impact of the branch instruction (the branch penalty).

- A several approaches have been taken for dealing with conditional branches:

+ Multiple streams

+ Prefetch branch target

+ Loop buffer

+ Branch prediction

+Delayed branch.

- Multiple streams

- Replicate the initial portions of the pipeline and fetch both possible next instructions

- Increases chance of memory contention

- Must support multiple streams for each instruction in the pipeline

- Prefetch branch target

- When the branch instruction is decoded, begin to fetch the branch target instruction and place in a second prefetch buffer

- If the branch is not taken, the sequential instructions are already in the pipe, so there is not loss of performance

- If the branch is taken, the next instruction has been prefetched and results in minimal branch penalty (don’t have to incur a memory read operation at the end of the branch to fetch the instruction)

- Loop buffer: Look ahead, look behind buffer

- Many conditional branches operations are used for loop control

- Expand prefetch buffer so as to buffer the last few instructions executed in addition to the ones that are waiting to be executed

- If buffer is big enough, entire loop can be held in it, this can reduce the branch penalty.

- Branch prediction

- Make a good guess as to which instruction will be executed next and start that one down the pipeline.

- Static guesses: make the guess without considering the runtime history of the program

Branch never taken

Branch always taken

Predict based on the opcode

- Dynamic guesses: track the history of conditional branches in the program.

Taken / not taken switch History table

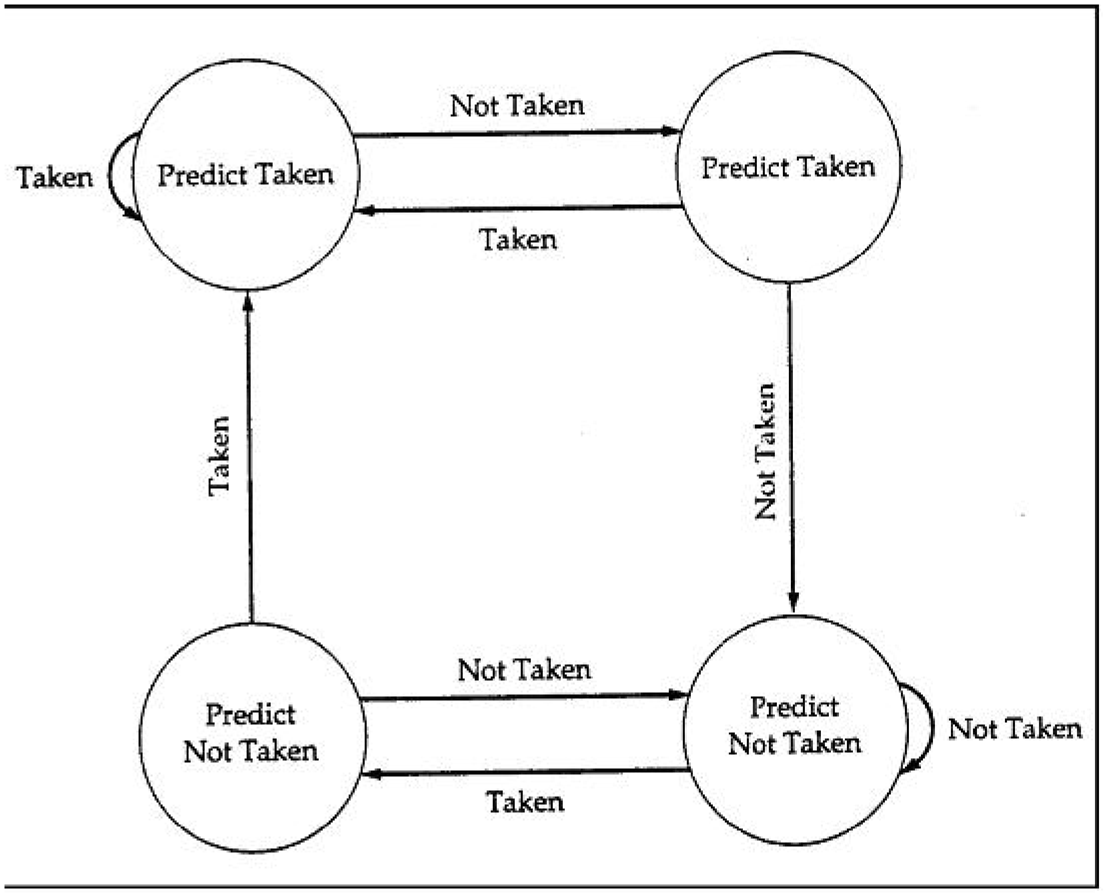


Figure 8.3. Branch prediction using 2 history bits

- Delayed branch

- Minimize the branch penalty by finding valid instructions to execute in the pipeline while the branch address is being resolved.

- It is possible to improve performance by automatically rearranging instruction within a program, so that branch instruction occur later than actually desired

- Compiler is tasked with reordering the instruction sequence to find enough independent instructions (wrt to the conditional branch) to feed into the pipeline after the branch that the branch penalty is reduced to zero

3. Superscalar and Superpipelined Processors

3.1 Superpipeline designs

– Observation: a large number of operations do not require the full clock cycle to complete

– High performance can be obtained by subdividing the clock cycle into a number of sub intervals » Higher clock frequency!

– Subdivide the “macro” pipeline H/W stages into smaller (thus faster) substages and clock data through at the higher clock rate

– Time to complete individual instructions does not change

» Degree of parallelism goes up

» Perceived speedup goes up

3.2 Superscalar

– Implement the CPU such that more than one instruction can be performed (completed) at a time

– Involves replication of some or all parts of the CPU/ALU

– Examples:

» Fetch multiple instructions at the same time

» Decode multiple instructions at the same time

» Perform add and multiply at the same time

» Perform load/stores while performing ALU operation

– Degree of parallelism and hence the speedup of the machine goes up as more instructions are executed in parallel

- Data dependencies in superscalar

– It must insure computed results are the same as would be computed on a strictly sequential machine

– Two instructions can not be executed in parallel if the (data) output of one is the input of the other or if they both write to the same output location

– Consider:

S1: A = B + C

S2: D = A + 1

S3: B = E + F

S4: A = E + 3

Resource dependencies:

– In the above sequence of instructions, the adder unit gets a real workout!

– Parallelism is limited by the number of adders in the ALU

3.3 Instruction issue policy

Problem: In what order are instructions issued to the execution unit and in what order do they finish?

There is 3 types of ordering.

- The order in which instructions are fetched

- The order in which instructions are executed

- The order in which instructions update the contents of registre or memory location.

- In-order issue, in-order completion

» Simplest method, but severely limits performance

» Strict ordering of instructions: data and procedural dependencies or resource conflicts delay all subsequent instructions

» Delay execution of some instructions delay all subsequent instructions

- In-order issue, out-of-order completion

» Any number of instructions can be executed at a time

» Instruction issue is still limited by resource conflicts or data and procedural dependencies

» Output dependencies resulting from out-of order completion must be resolved

» “Instruction” interrupts can be tricky

- Out-of-order issue, out-of-order completion

» Decode and execute stages are decoupled via an instruction buffer “window”

» Decoded instructions are “stored” in the window awaiting execution

» Functional units will take instructions from the window in an attempt to stay busy

This can result in out-of-order execution

S1: A = B + C

S2: D = E + 1

S3: G = E + F

S4: H = E * 3

“Antidependence” class of data dependencies must be dealt with it.

Module 9: Multilevel Memories

1. Computer Memory System and characteristics

1.1 Computer Memory Overview

The memory is that part of computer where programs and data are stored. The basical concept is the following:

- Bits

The basic unit of memory is the binary digit called a bit. A bit may contain a 0 or 1. It is the simplest possible unit

- Memory addresses

- Memories consist of a number of cells or locations each of which can store a piece of information. Each location has a number called its address, by which program can refer to it. The cells is the smallest addressable

- If an address has m bits, the maximum number of cells addressable is 2m.

- Byte: 8-bits

- Bytes are grouped into words. The significance of word is that most instruction operate on entire word. A computer with a 32bit/word has 4 bytes/word

- Byte ordering

- The bytes in a word can be numbered from left-to-right or right-to-left.

- The former system, where the numbering begin at the "big" (i.e, high-order) end is called a big endian computer, such as the SPARC or the big IBM mainframes. In contras it is a little endian computer, such as the Intel family using right-to-left numbering for the representation of a 32 bit computer.

1.2 Characteristics of Memory System

- Capacity: the amount of information that can be contained in a memory unit -- usually in terms of words or bytes
- Memory word: the natural unit of organization in the memory, typically the number of bits used to represent a number
- Addressable unit: the fundamental data element size that can be addressed in the memory -- typically either the word size or individual bytes
- Unit of transfer: The number of data elements transferred at a time – usually bits in main memory and blocks in secondary memory
- Transfer rate: Rate at which data is transferred to/from the memory device
- Access time:

– For RAM, the time to address the unit and perform the transfer

– For non-random access memory, the time to position the R/W head over the desired location

- Memory cycle time: Access time plus any other time required before a second access can be started
- Access technique: how are memory contents accessed

– Random access:

» Each location has a unique physical address

» Locations can be accessed in any order and all access times are the same

» What we term “RAM” is more aptly called

read/write memory since this access technique also applies to ROMs as well

» Example: main memory

– Sequential access:

» Data does not have a unique address

» Must read all data items in sequence until the desired item is found

» Access times are highly variable

» Example: tape drive units

– Direct access:

» Data items have unique addresses

» Access is done using a combination of moving to a general memory “area” followed by a sequential access to reach the

desired data item

» Example: disk drives

– Associative access:

» A variation of random access memory

» Data items are accessed based on their contents rather than their actual location

» Search all data items in parallel for a match to a given search pattern

» All memory locations searched in parallel without regard to the size of the memory

» Extremely fast for large memory sizes

» Cost per bit is 5-10 times that of a “normal” RAM cell

» Example: some cache memory units.

2. Types of Memory

Computer memory system consists a various types of memory. Manufactures produce a number of different types of memory devices having a variety of technologies. The technology affect not only the operating chracteristics but also the manufacturing cost. In the section following we present an overviews of types of memory. You can see the study in detail of memory in the modules 10, 11 and 12.

- Main Memory (“Internal” memory components)

- RAM (read-write memory): Static RAM, Dynamic RAM

- ROM (Read Only Memories) : ROMs, PROMs, EPROMs, EEPROMs, Flash Memory.

- Cache memory

The cache memories are high-speed buffers for holding recently accessed data and neighboring data in main memory. The organization and operations of cache provide an apparently fast memory system.

- External Memory

- Magnetic disks

- RAID technology disks

- Optical disks

- Magnetic tape

3. Memory Hierarchy

3.1 Memory System Organization

No matter how big the main memory, how we can organize effectively the memory system in order to store more information than it can hold. The traditional solution to storing a great deal of data is a memory hierarchy.

- Major design objective of any memory system:

– To provide adequate storage capacity at

– An acceptable level of performance

– At a reasonable cost

- Four interrelated ways to meet this goal

– Use a hierarchy of storage devices

– Develop automatic space allocation methods for efficient use of the memory

– Through the use of virtual memory techniques, free the user from memory management tasks

– Design the memory and its related interconnection structure so that the proces

3.2 Multilevel Memories Organization

Three key characteristics increase for a memory hierarchy. They are the access time, the storage capacity and the cost. The memory hierarchy is illustrated in figure 9.1.

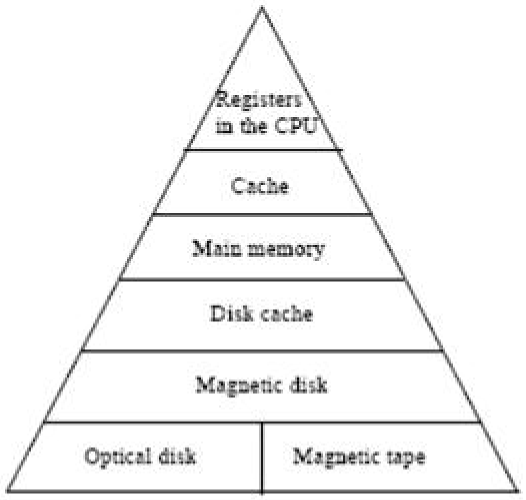


Figure 9.1. The memory hierarchy

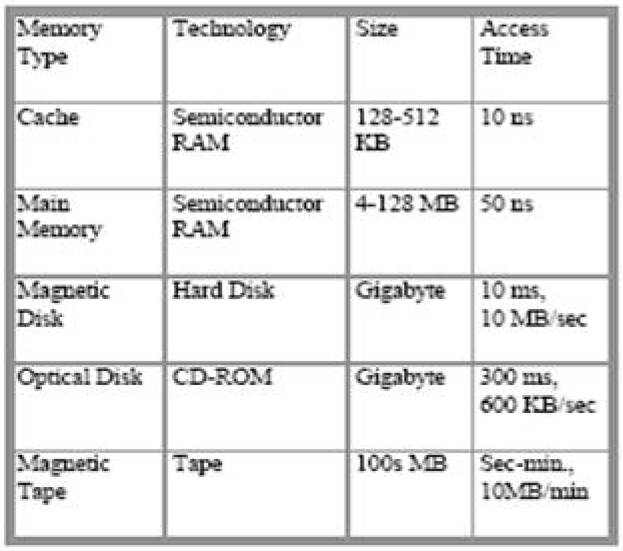
We can see the memory hierarchy with six levels. At the top there are CPU registers, which can be accessed at full CPU speed. Next commes the cache memory, which is currently on order of 32 KByte to a few Mbyte. The main memory is next, with size currently ranging from 16 MB for entry-level systems to tens of Gigabytes. After that come magnetic disks, the current work horse for permanent storage. Finally we have magnetic tape and optical disks for archival storage.

- Basis of the memory hierarchy

– Registers internal to the CPU for temporary data storage (small in number but very fast)

– External storage for data and programs (relatively large and fast)

– External permanent storage (much larger and much slower)



Memory Type	Technology	Size	Access Time
Cache	Semiconductor RAM	128-512 KB	10 ns
Main Memory	Semiconductor RAM	4-128 MB	50 ns
Magnetic Disk	Hard Disk	Gigabyte	10 ms, 10 MB/sec
Optical Disk	CD-ROM	Gigabyte	300 ms, 600 KB/sec
Magnetic Tape	Tape	100s MB	Sec-min., 10MB/min

Figure 9.2 Typical Memory Parameters

- Characteristics of the memory hierarchy

– Consists of distinct “levels” of memory components

– Each level characterized by its size, access time, and cost per bit

– Each increasing level in the hierarchy consists of modules of larger capacity, slower access time, and lower cost/bit

4. Memory Performance

Goal of the memory hierarchy. Try to match the processor speed with the rate of information transfer from the lowest element in the hierarchy.

- The memory hierarchy speed up the memory performance

The memory hierarchy works because of locality of reference

– Memory references made by the processor, for both instructions and data, tend to cluster together

+ Instruction loops, subroutines

+ Data arrays, tables

– Keep these clusters in high speed memory to reduce the average delay in accessing data

– Over time, the clusters being referenced will change -- memory management must deal with this

- Performance of a two level memory

Example: Suppsose that the processor has access to two level of memory:

– Two-level memory system

– Level 1 access time of 1 us

– Level 2 access time of 10us

– Ave access time = H(1) + (1-H)(10) ns

where: H is a fraction of all memory access that are found in the faster memory (e.g cache)

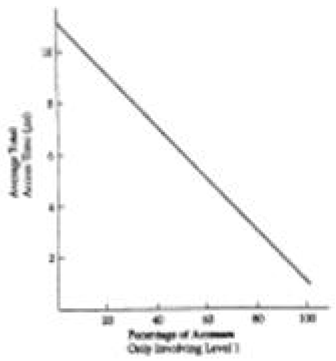


Figure 9.3. Performance of a two level memory

Module 10: Cahe Memory

1. Cache Memory Overview

Cache memory is the fast and small memory. The basis idea of using a cache is simple. When CPU need a word, it first looks in the cache. Only if the word is not there does it go to main memory. If a substantial fraction of the words are in the cache, the average access time can be greatly reduced. So the success or faillure thus depends on what fraction of the words are in cache. The idea solution is that when a word is referenced, it and some of its neighbors are brought from the large slow memory into the cache, so that the next time it can be accessed quickly.

- Improving Performance

The memory mean access time, TM, is considered as follow:

$TM = c + (1- h).m$

Where:

c : the cache access time

m: the main memory access time

h: the hit ratio, which is the fraction of all references that can be satisfied out of cache (the success of cache), also the miss ratio is 1-h.

- Cache memory is a critical component of the memory hierarchy.

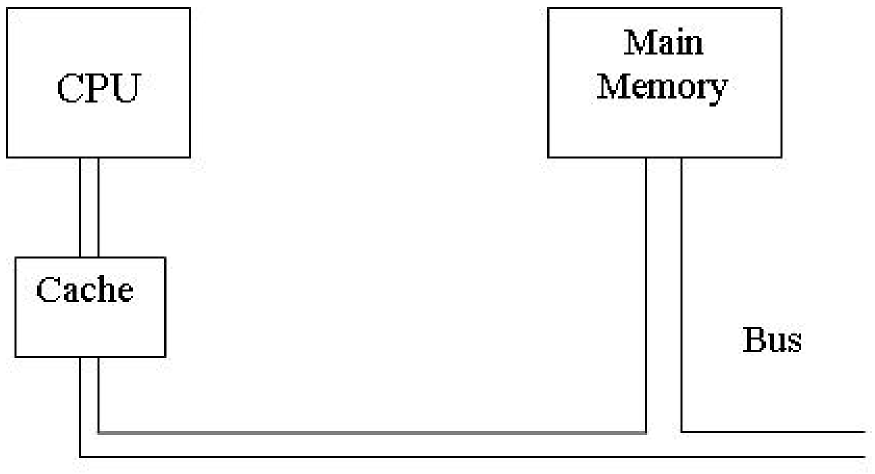


Figure 10.1. The cache is logically between the CPU and main memory in the the memory hierarchy. Physically, there are several possible places it could be located.

- Characteristics

– Compared to the size of main memory, cache is relatively small

– Operates at or near the speed of the processor

– Cache is very expensive compared to main memory

– Cache contains copies of sections of main memory

- Cache is considered as the main memory interface

– Assume an access to main memory causes a block of K words to be transferred to the cache

– The block transferred from main memory is stored in the cache as a single unit called a slot, line, or page

– Once copied to the cache, individual words within a line can be accessed by the CPU

– Because of the high speeds involved with the cache, management of the data transfer and storage in the cache is done in hardware

- If there are 2n words in the main memory, then there will be M=2n /K blocks in the memory

– M will be much greater than the number of lines, C, in the cache

– Every line of data in the cache must be tagged in some way to identify what main memory block it is. The line of data and its tag are stored in the cache.

- Factors in the cache design

- Mapping function between main memory and the cache

- Line replacement algorithm

- Write policy

- Block size

- Number and type of caches

2. Cache Organization and Operations

Mapping function organization: Mapping functions since M>>C, how are blocks mapped to specific lines in cache

2.1 Direct mapping

a/ Organization:

The simplest cache is known as a direct-mapped cache or direct mapping, it is shown in Figure 10.2.

– Direct mapping cache treats a main memory address as 3 distinct fields

+ Tag identifier

+ Line number identifier

+ Word identifier (offset)

– Word identifier specifies the specific word (or addressable unit) in a cache line that is to be read

– Line identifier specifies the physical line in cache that will hold the referenced address

– The tag is stored in the cache along with the data words of the line

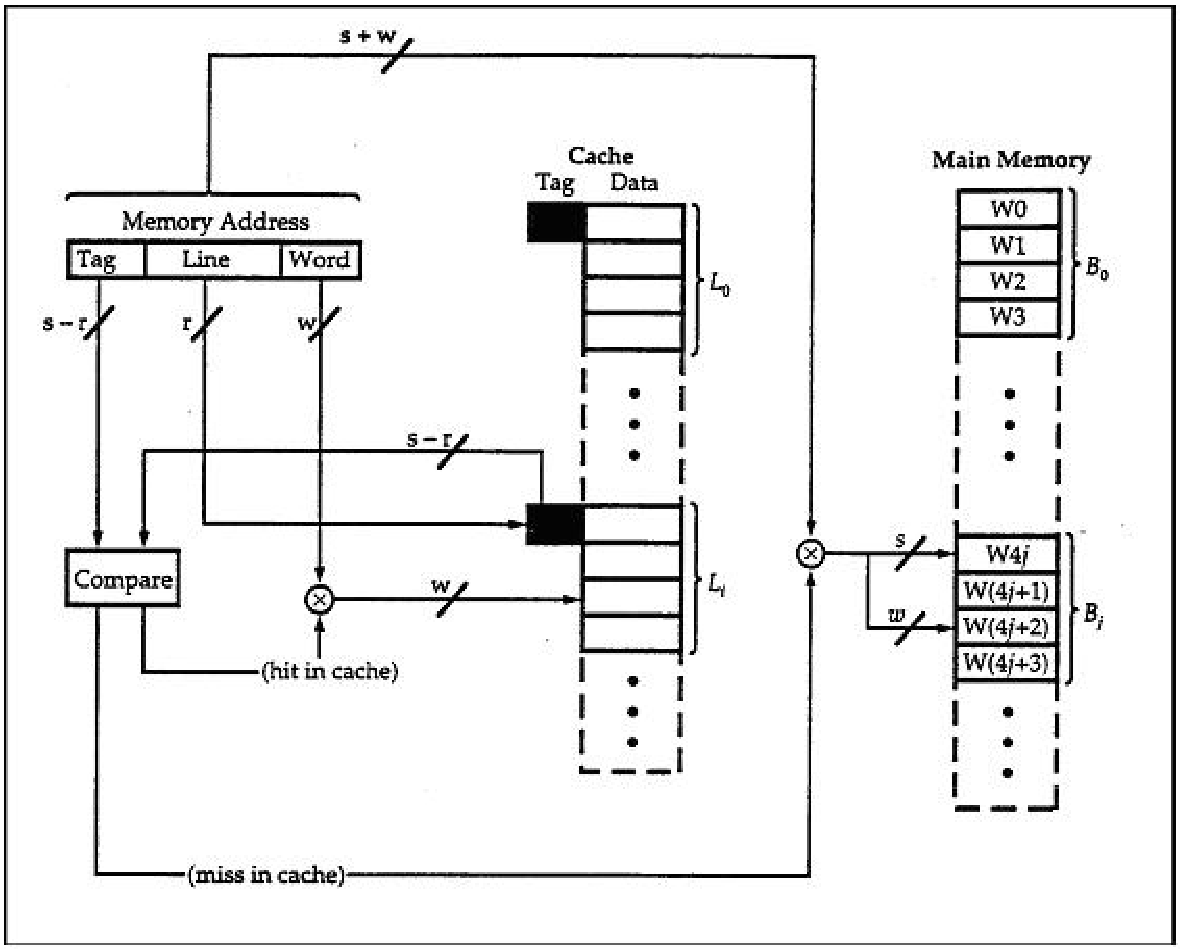


Figure 10.2. Direct mapping cache Organization

For every memory reference that the CPU makes, the specific line that would hold the reference (if it is has already been copied into the cache) is determined. The tag held in that line is checked to see if the correct block is in the cache

b/ Operations

- Each main memory block is assigned to a specific line in the cache:

i = j modulo C, where i is the cache line number assigned to main memory block j

– If M=64, C=4:

Line 0 can hold blocks 0, 4, 8, 12, ...

Line 1 can hold blocks 1, 5, 9, 13, ...

Line 2 can hold blocks 2, 6, 10, 14, ...

Line 3 can hold blocks 3, 7, 11, 15, ...

– Example:

Memory size of 1 MB (20 address bits) addressable to the individual byte

Cache size of 1 K lines, each holding 8 bytes:

Word id = 3 bits

Line id = 10 bits

Tag id = 7 bits

Where is the byte stored at main memory

location $ABCDE stored?

$ABCDE=1010101 1110011011 110

Cache line $39B, word offset $6, tag $55

c/ Remarks

- Advantages of direct mapping

+ Easy to implement

+ Relatively inexpensive to implement

+ Easy to determine where a main memory reference can be found in cache

- Disadvantage

+ Each main memory block is mapped to a specific cache line

+ Through locality of reference, it is possible to repeatedly reference to blocks that map to the same line number

+ These blocks will be constantly swapped in and out of cache, causing the hit ratio to be low.

2.2 Associative mapping

a/ Organization

A set-associative cache or associative mapping is inhenrently more complicated than a direct-mapped cache because although the correct cache entry to examine can be computed from the memory address being referenced, a set of n cache entries must be checked to see if the need line is present.

Associative mapping can overcomes direct mapping’s main of the direct mapping, the associate cache organization is illustrated in Figure 10.3.

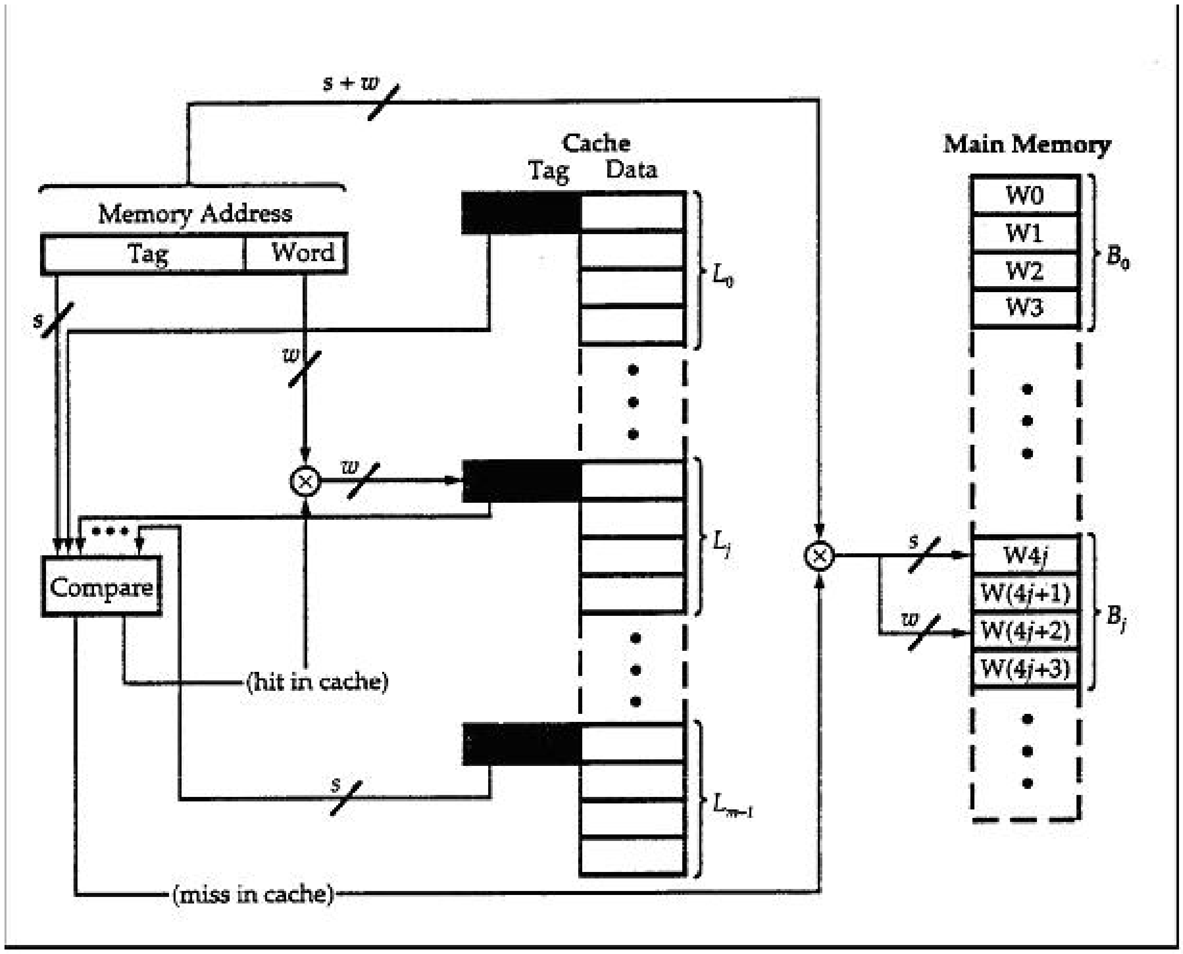


Figure 10.3. Associate Cache Organization

– Operation must examine each line in the cache to find the right memory block

+ Examine the line tag id for each line

+ Slow process for large caches!

– Line numbers (ids) have no meaning in the cache

+ Parse the main memory address into 2 fields (tag and word offset) rather than 3 as in direct mapping

– Implementation of cache in 2 parts:

+ Part SRAM: The lines themselves in SRAM

+ The tag storage in associative memory

– Perform an associative search over all tags

b/ Operation example

With the same example: Memory size of 1 MB (20 address bits) addressable to the individual byte. Cache size of 1 K lines, each holding 8 bytes:

Word id = 3 bits

Tag id = 17 bits

Where is the byte stored at main memory location $ABCDE stored?

$ABCDE=10101011110011011 110

Cache line unknown, word offset $6, tag $1579D.

c/ Remarks

- Advantages

- Fast

- Flexible

- Disadvantage
- Implementation cost

Example above has 8 KB cache and requires 1024 x 17 = 17,408 bits of

associative memory for the tags!

2.3 Set associative mapping

a/ Organization

The associative mapping is considered as compromise between direct and fully associative mappings that builds on the strengths of both

– Divide cache into a number of sets (v), each set holding a number of lines (k)

– A main memory block can be stored in any one of the k lines in a set such that

set number = j modulo v

– If a set can hold X lines, the cache is referred to as an X-way set associative cache

Most cache systems today that use set associative mapping are 2- or 4-way set

Associative.

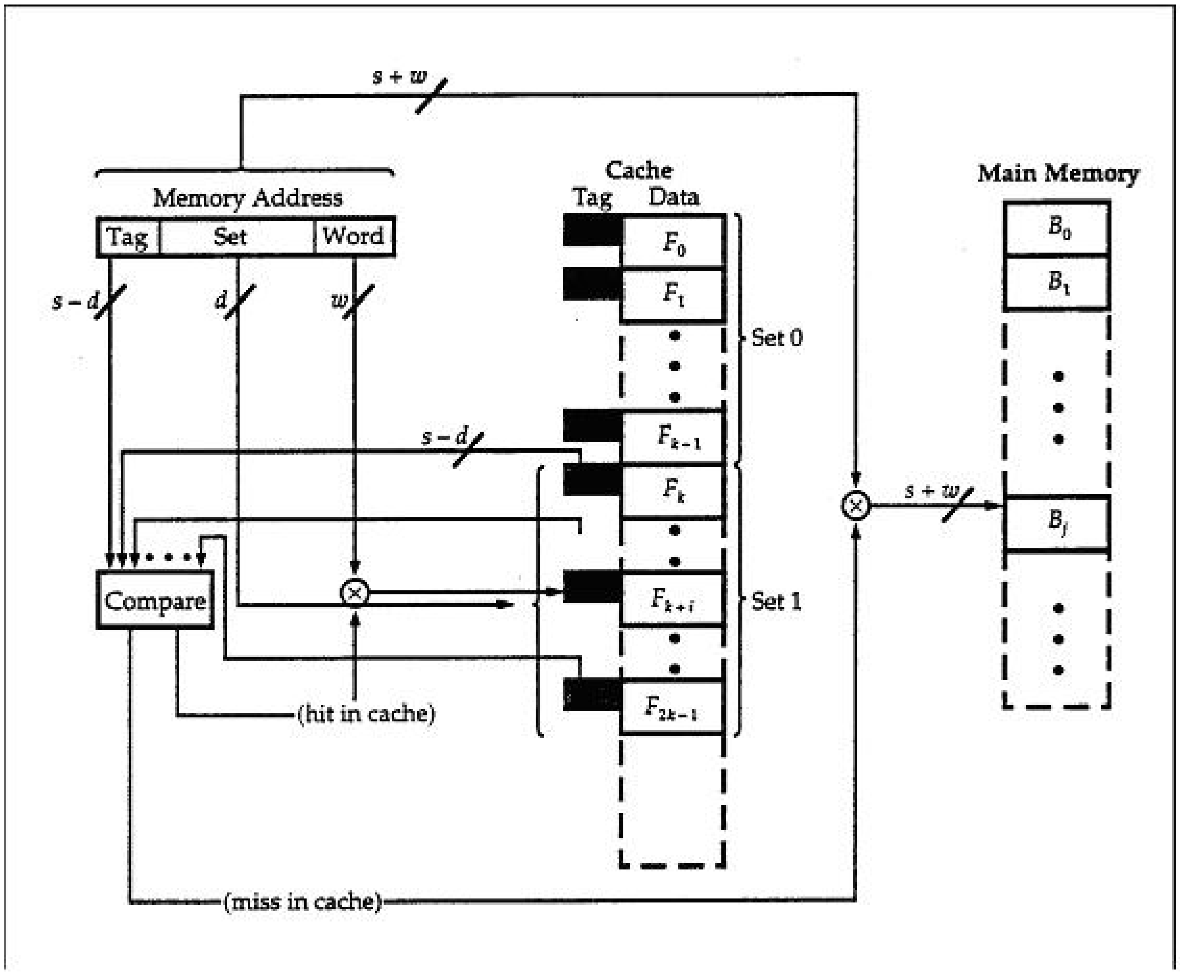


Figure. 10. 4 Set associative cache organization

b/ Example

Assume the 1024 lines are 4-way set associative:

1024/4 = 256 sets

Word id = 3 bits

Set id = 8 bits

Tag id = 9 bits

Where is the byte stored at main memory

Location $ABCDE stored?

$ABCDE=101010111 10011011 110

Cache set $9B, word offset $6, tag $157

3. Replacement Algorithms

As we known, cache is the fast and small memory. when new block is brough into the cache, one of the blocks existing must be replaced by the new block that is to be read from memory.

For direct mapping, there is only one possible line for any particular block and no choice is possible. For the associative cache or a set associative cache, a replacement algorithm is needed.

A number of algorithms can be tried:

– Least Recently Used (LRU)

– First In First Out (FIFO)

– Least Frequently Used (LFU)

– Random

Probably the most effective algorithm is least recently used (LRU): Replace that block in the set that has been in the cache longest with no refrence.

4. Performances

4.1 Write policy

– When a line is to be replaced, must update the original copy of the line in main memory if any addressable unit in the line has been changed

– Write through

+ Anytime a word in cache is changed, it is also changed in main memory

+ Both copies always agree

+ Generates lots of memory writes to main memory

– Write back

+ During a write, only change the contents of the cache

+ Update main memory only when the cache line is to be replaced

+Causes “cache coherency” problems -- different values for the contents of an address are in the cache and the main memory

+ Complex circuitry to avoid this problem

4.2 Block / line sizes

– How much data should be transferred from main memory to the cache in a single memory reference

– Complex relationship between block size and hit ratio as well as the operation of the system bus itself

– As block size increases,

+ Locality of reference predicts that the additional information transferred will likely be used and thus increases the hit ratio (good)

+ Number of blocks in cache goes down, limiting the total number of blocks in the cache (bad)

+ As the block size gets big, the probability of referencing all the data in it goes down (hit ratio goes down) (bad)

+ Size of 4-8 addressable units seems about right for current systems

Number of caches

Single vs. 2-level cache:

- Modern CPU chips have on-board cache (Level 1 – L1):

L1 provides best performance gains

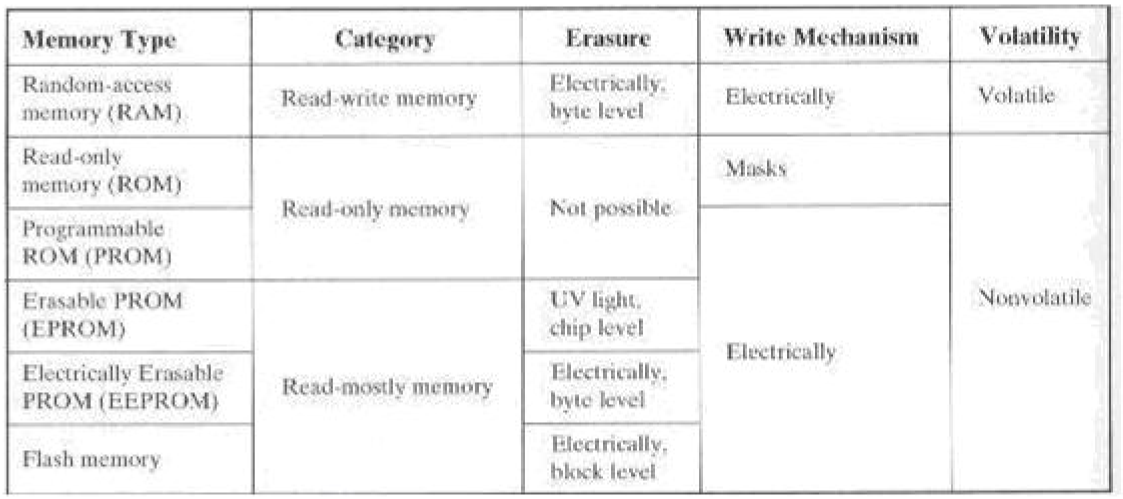
- Secondary, off-chip cache (Level 2) provides higher speed access to main memory

L2 is generally 512KB or less more than this is not cost-effective.

Module 11: Internal Memory

1. Semiconductor Main Memory

The basic element of a semiconductor memory is the memory cell. There are a lot of semiconductor memory types shown in table 11.1



Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip level		
Electrically Erasable PROM (EEPROM)		Electrically, byte level		
Flash memory		Electrically, block level		

Table 11.1 Semiconductor Memory Types

RAM: Read-write memory

The two basic forms of semiconductor Random Access Memory (RAM) are dynamic RAM (DRAM) and static RAM (SRAM). SRAM is faster, more expensive and less dense than DRAM. DRAM is usually used for main memory.

- Dynamic RAM

- Storage cell is essentially a transistor acting as a capacitor

- Capacitor charge dissipates over time causing a 1 to flip to a zero

- Cells must be refreshed periodically to avoid this

- Very high packaging density

- Static RAM: It is basically an array of flip-flop storage cells

- Uses 5-10x more transistors than similar dynamic cell so packaging density is 10x lower

- Faster than a dynamic cell

ROM : Read Only Memories

A Read only Memory (ROM) contain a permanent of data that canot be changed.

- “Permanent” data storage

- ROMs: Data is “wired in” during fabrication at a chip manufacturer’s plant

- Purchased in lots of 10k or more

- PROMs: Programmable ROM

- Data can be written once by the user employing a PROM programmer

- Useful for small production runs

- EPROM: Erasable PROM

- Programming is similar to a PROM

- Can be erased by exposing to UV light

- EEPROMS: Electrically erasable PROMs

- Can be written to many times while remaining in a system

- Does not have to be erased first

- Program individual bytes

- Writes require several hundred usec per byte

- Used in systems for development, personalization, and other tasks requiring unique information to be stored

- Flash Memory: Similar to EEPROM in using electrical erase

- Fast erasures, block erasures

- Higher density than EEPROM

2. Memory Organization

2.1 Memory Organization from the memory chip

Each memory chip contains a number of 1-bit cells. The 1, 4, and 16 million cell chips are common. The cells can be arranged as a single bit column (e.g., 4Mx1) or in multiple bits per address location (e.g., 1Mx4)

- To reduce pin count, address lines can be multiplexed with data and/or as high and low halves Trade off is in slower operation

- Typical control lines:

W* (write), OE* (output enable) for write and read operations

CS* (chip select) derived from external address decoding logic

RAS*, CAS* (row and column address selects) used when address is applied to the chip in 2 halves

- Example:

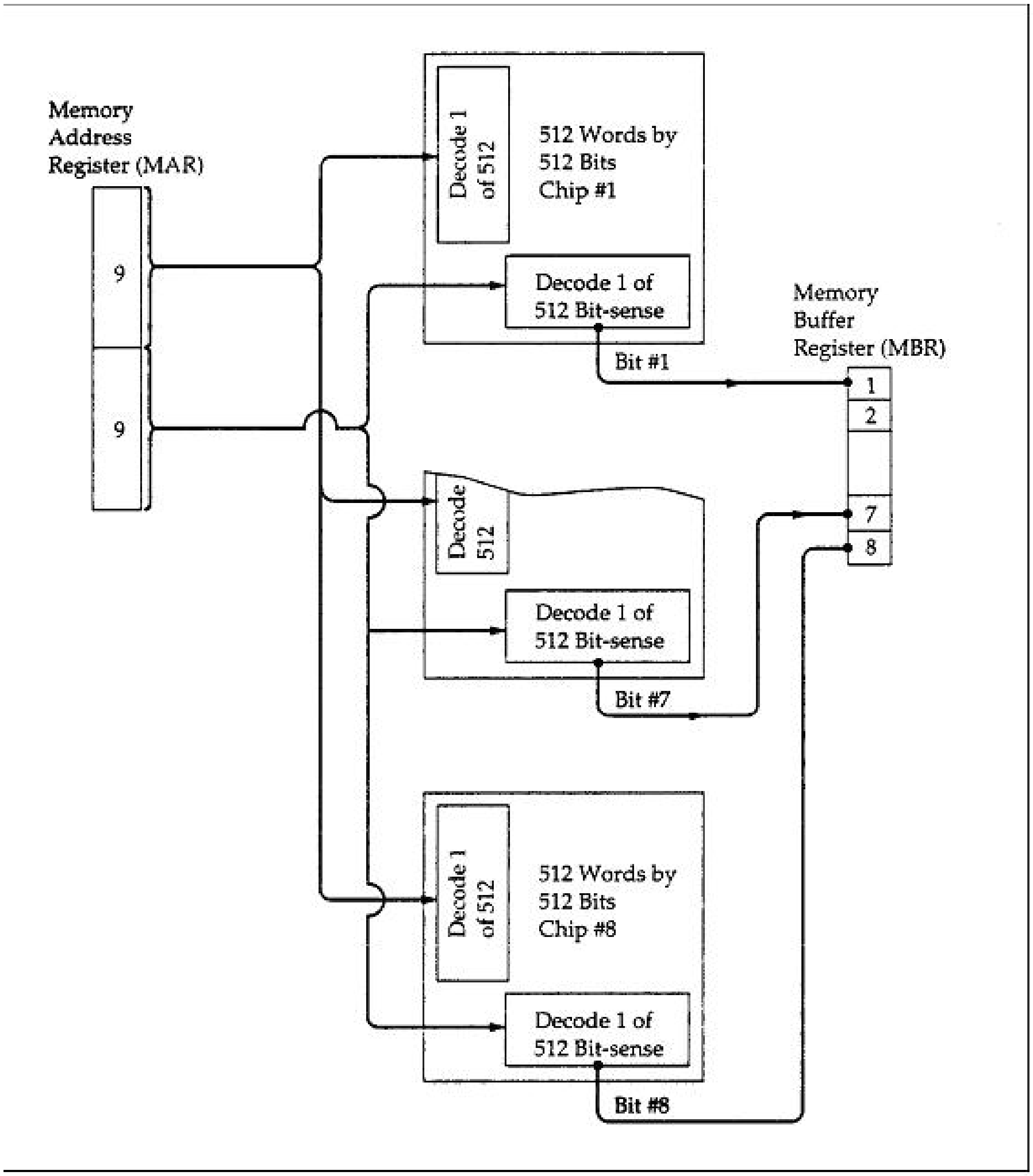

Memory Address Register (MAR)
9
9
Decode 1 of 512
512 Words by 512 Bits Chip #1
Decode 1 of 512 Bit-sense
Bit #1
Memory Buffer Register (MBR)
1
2
7
8
Decode 512
Decode 1 of 512 Bit-sense
Bit #7
Decode 1 of 512
512 Words by 512 Bits Chip #8
Decode 1 of 512 Bit-sense
Bit #8

Figure 11.1 Organization 256Kx8 memory from 256Kx1 chips

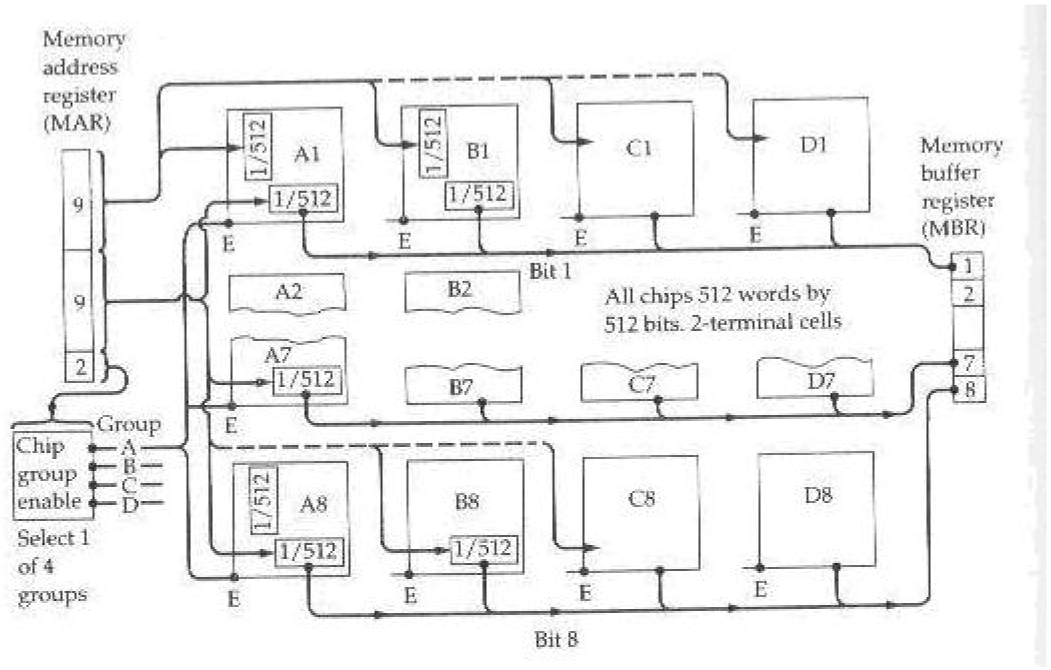

Memory address register (MAR)
9
9
2
A1
B1
C1
D1
1/512
E
Bit 1
A2
B2
All chips 512 words by 512 bits. 2-terminal cells
A7
B7
C7
D7
Memory buffer register (MBR)
1
2
7
8
Group
Chip group enable
A
B
C
D
Select 1 of 4 groups
A8
B8
C8
D8
Bit 8

Figure 11.2 . Mbyte Memory Organization

2.2 Error correction

- Problem: Semiconductor memories are subject to errors

– Hard (permanent) errors

» Environmental abuse

» Manufacturing defects

» Wear

– Soft (transient) errors

» Power supply problems

» Alpha particles: Problematic as feature sizes shrink

– Memory systems include logic to detect and/or correct errors

» Width of memory word is increased

» Additional bits are parity bits

» Number of parity bits required depends on the level of detection and correction needed

- General error detection and correction

– A single error is a bit flip -- multiple bit flips can occur in a word

– 2M valid data words

– 2M+K codeword combinations in the memory

– Distribute the 2M valid data words among the 2 M+K codeword combinations such that the “distance” between valid words is sufficient to distinguish the error

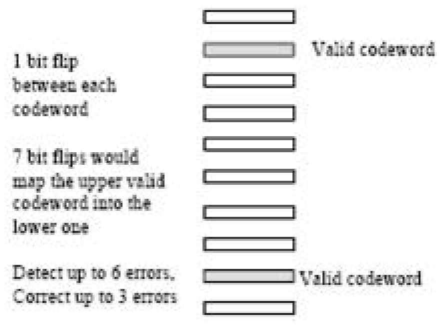

Valid codeword
1 bit flip between each codeword
7 bit flips would map the upper valid codeword into the lower one
Detect up to 6 errors, Correct up to 3 errors
Valid codeword

Figure 11.3

- Single error detection and correction

– For each valid codeword, there will be 2K-1 invalid codewords

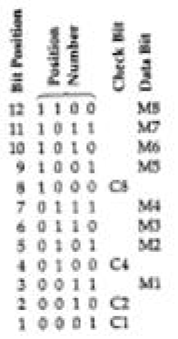
– 2K-1 must be large enough to identify which of the M+K bit positions is in error

– Therefore 2K-1 > M+K

» 8-bit data, 4 check bits

» 32-bit data, 6 check bits

– Arrange bits as shown in Figure 11.4



Bit Position	Position Number				Check Bit	Data Bit
12	1	1	0	0		M8
11	1	0	1	1		M7
10	1	0	1	0		M6
9	1	0	0	1		M5
8	1	0	0	0	C8	
7	0	1	1	1		M4
6	0	1	1	0		M3
5	0	1	0	1		M2
4	0	1	0	0	C4	
3	0	0	1	1		M1
2	0	0	1	0	C2	
1	0	0	0	1	C1	

Figure 11.4

– Bit position n is checked by bits Ci such that the sum of the subscripts, i, equals n (e.g., position 10, bit M6, is checked by bits C2 and C8)

To detect errors, compare the check bits read from memory to those computed during the read operation (use XOR)

+ If the result of the XOR is 0000, no error

+ If non-zero, the numerical value of the result indicates the bit position in error

+ If the XOR result was 0110, bit position 6 (M3) is in error

Double error detection can be added by adding another check bit that implements a parity check for the whole word of M+K bits. SED and SEC-DED are generally enough protection in typical systems

Module 12: External Memory

1. Magnetic disks

1.1 Magnetic disks overview

The magnetic disks are the foundation of external memory on virtually all computer system. Both removable and fixed disk or hard disk are used in computer system from personal computer to mainframe or supercpomputer.

- Principle

– The disk is a metal or plastic platter coated with magnetizable material

– Data is recorded onto and later read from the disk using a conducting coil, the head

– Data is organized into concentric rings, called tracks, on the platter

– Tracks are separated by gaps

– Disk rotates at a constant speed – constant angular velocity

The number of data bits per track is a constant

The data density is higher on the inner tracks

– Logical data transfer unit is the sector

Sectors are identified on each track during the formatting process

- Disk characteristics

- Single vs. multiple platters per drive (each platter has its own read/write head)

- Fixed vs. movable head. Fixed head has a head per track. Movable head uses one head per platter

- Removable vs. nonremovable platters. Removable platter can be removed from disk drive for storage of transfer to another machine

- Data accessing times:

+ Seek time -- position the head over the correct track

+ Rotational latency -- wait for the desired sector to come under the head

+ Access time -- seek time plus rotational latency

+ Block transfer time -- time to read the block (sector) off of the disk and

transfer it to main memory.

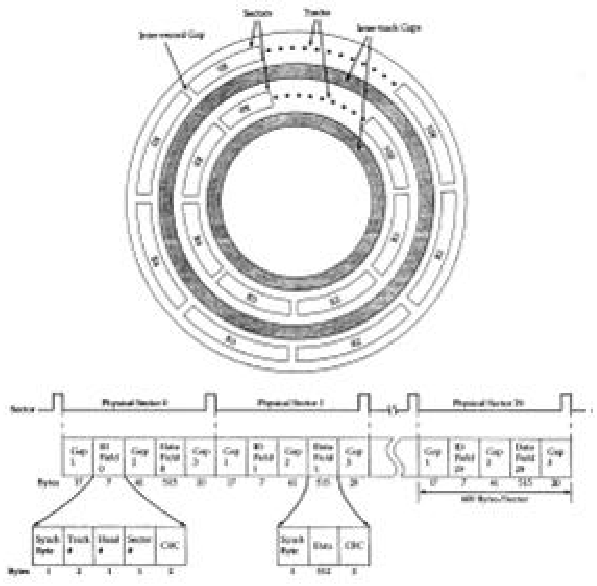


Figure 12.1. Disk Organization

1.2 RAID Technology

The RAID (Redundant Array of Independent Disk) technology can obtain greater performance and higher availability. RAID refers to a family of techniquesfor using multiple disks as a paralle array of data storage devices with redundant built in to compensate for disk failure.

– Disk drive performance has not kept pace withimprovements in other parts of the system

– Limited in many cases by the electromechanical transport means

– Capacity of a high performance disk drive can be duplicated by operating many (much cheaper) disks in parallel with simultaneous access

– Data is distributed across all disks

– With many parallel disks operating as if they were a single unit, redundancy techniques can be used to guard against data loss in the unit (due to aggregate failure rate being higher)

– “RAID” developed at Berkeley – Redundant Array of Independent Disks

- Six levels RAID: 0 – 5

– RAID 0

» No redundancy techniques are used

» Data is distributed over all disks in the array

» Data is divided into strips for actual storage similar in operation to interleaved

memory data storage

» RAID can be used to support high data transfer rates by having block transfer size be in multiples of the strip

» RAID can support low response time by having the block transfer size equal a strip --

support multiple strip transfers in parallel

– RAID 1

All disks are mirrored – duplicated Data is stored on a disk and its mirror Read from either the disk or its mirror Write must be done to both the disk and mirror

- Fault recovery is easy -- use the data on the mirror
- System is expensive!

– RAID 2

All disks are used for every access – disks are synchronized together

Data strips are small (byte)

Error correcting code computed across all disks and stored on additional disks

Uses fewer disks than RAID 1 but still expensive.Number of additional disks is

proportional to log of number of data disks

– RAID 3

Like RAID 2 however only a single redundant disk is used -- the parity drive

Parity bit is computed for the set of individual bits in the same position on all

Disks If a drive fails, parity information on the redundant disks can be used to calculate thedata from the failed disk “on the fly”

2. Optical disks

Advent of CDs in the early 1980s revolutionized the audio and computer industries

- Basic operations

» CD is operated using constant linear velocity

» Essentially one long track spiraled onto the disk

» Track passes under the disk’s head at a constant rate -- requires the disk to change rotational speed based on what part of the track you are on

» To write to the disk, a laser is used to burn pits into the track -- write once!

» During reads, a low power laser illuminates the track and its pits

- In the track, pits reflect light differently than no pits thus allowing you to store 1s and 0s

– Characteristics:

- Master disk is made using the laser

- Master is used to “press” copies in a mass production mechanical style

- Cheaper than production of information on magnetic disks

- Storage capacity roughly 775 NB or 550 3.5” disks

- Transfer rate standard is 176 MB/second

- Only economical for production of large quantities of disks

- Disks are removable and thus archival

- Slower than magnetic disks.

- WORMs -- Write Once, Read Many disks

» User can produce CD ROMs in limited quantities

» Specially prepared disk is written to using a medium power laser

» Can be read many times just like a normal CD ROM

» Permits archival storage of user information, distribution of large amounts of information by a user.

- Erasable optical disk

» Combines laser and magnetic technology to permit information storage

» Laser heats an area that can then have an efield orientation changed to alter information storage

» “State of the e-field” can be detected using polarized light during reads.

3. Magnetic Tape

– The first kind of secondary memory

– Still widely used

» Very cheap

» Very slow

– Sequential access

» Data is organized as records with physical air gaps between records

» One words is stored across the width of the tape and read using multiple read/write heads.

Module 13: Input/Output

1. I/O Overview

The computer system's I/O architecture is its interface to the outside world. The I/O components consist the buses, the keybroads, CRT Minitor, Flat panel display, scanner, modem …The I/O architecture consiste the buses and a set of I/O modules, each module interfaces to the bus system or to the outside world.

- There are 3 principal I/O techniques:

- Programmed I/O

- Interrupt-dreiven I/O

- Direct Memory Access (DMA)

- There are two I/O techniques.

I/O parallel

I/O serial

1. Buses

The collection of paths that connect the system modules together form the interconnection structure.

- Bus Interconnection

Computer systems contain a number of buses that provide pathways between

components

– Shared transmission media connecting 2 or more devices together

– Broadcast, 1-to-all operation

– Must insure only 1 device places information onto a bus at any given time

- Typical buses consist of 50-100 lines

- Address information (address bus)

Specifies source/destination of data transfer

Width determines the capacity of the system

– Data information (data bus)

Width is key in determining overall performance

– Control information: Controls access to and use of the address and data bus

– Miscellaneous: power, ground, clock

- Bus performance and limitations

The performance is limited by:

+ Data propagation delay through the bus longer buses (to support more devices) require longer delays

+ Aggregate demand for access to the bus from all devices connected to the bus

multiple buses To avoid bottlenecks,

- Multiple Buses:

– Hierarchical

– High-speed limited access buses close to the processor

– Slower-speed general access buses farther away from the processor.

- External Bus - PC bus

ISA (Industrial Standard Architecture)

– First open system bus architecture for PCs (meaning IBM-type machines)

– 8-bit and 16-bit ISA buses

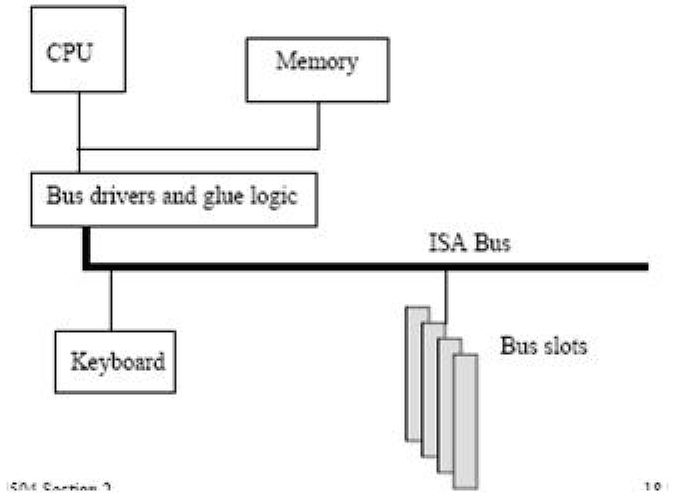


Figure 13.2 ISA Bus

- Micro Channel Architecture

- PCI

- VESA Video Local Bus
- Bus architecture and organization

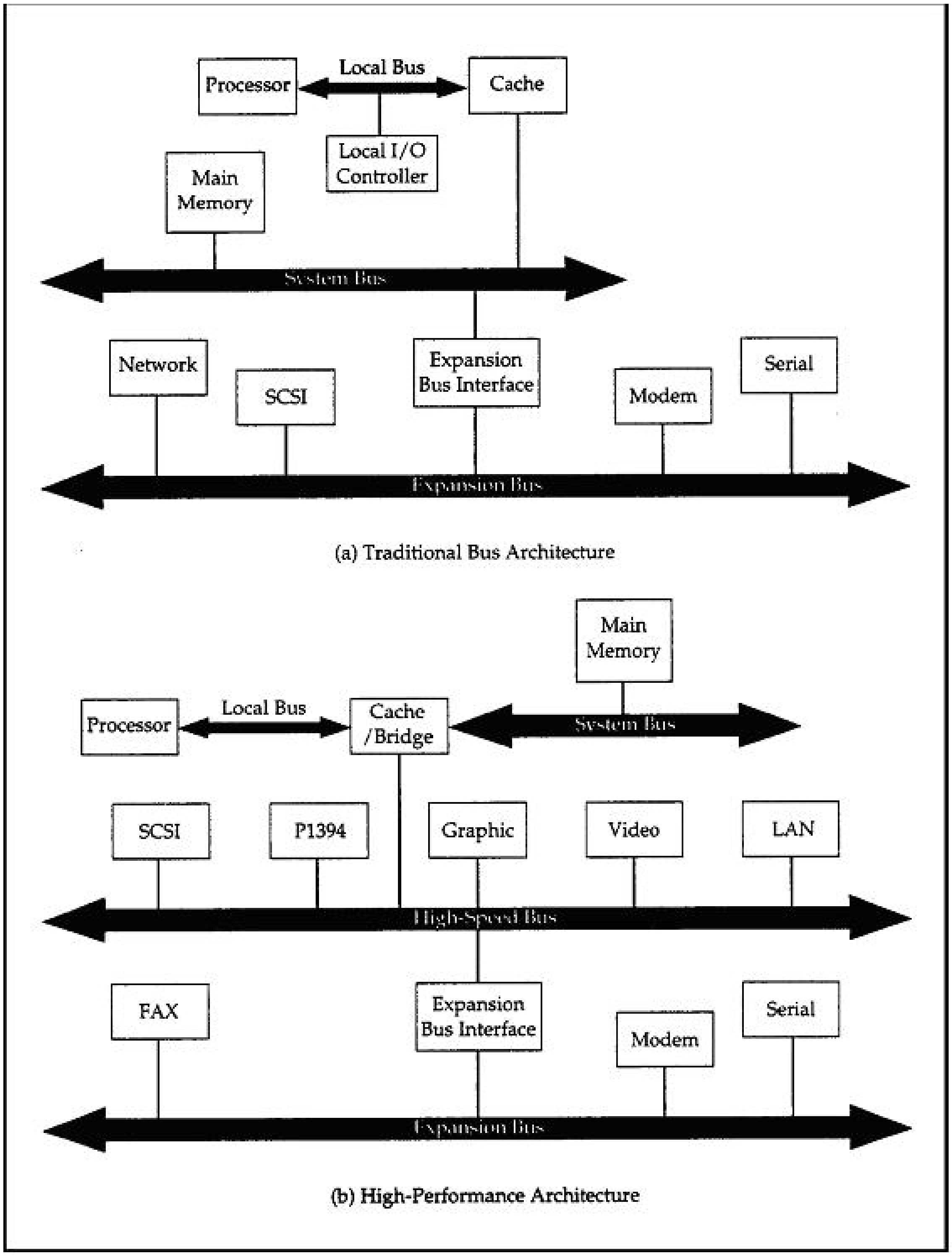


Figure 13.1 Hierarchical bus configurations

I/O Modules and InterfaceStructure

Programmed I/O

Interrup Drive I/O

Direct Memory Access

Module 14: Operating System Support

1. Introduction to Operating System (OS)

The Operating System is the software that control the execution of programs on a processor and that manages the computer's resources

- Operating System

– The Operating System (OS) is the program that manages the resources, provides

services to programmer, schedules execution of other programs.

– The OS masks the details of the hardware from the programmer

– The OS provides programmer a good interface for using the system that consists the new instructions are called system calls

– The OS mediates programmer and application programs requests for facilities and services

- Services provided

– Program creation -- general utilities

– Program execution -- loading, device

initializing, etc.

– Standardized access to I/O devices

– Controlled access to files

– Overall system access control

- Types of Operating System

– Interactive OS

User directly interacts with the OS through a keyboard / terminal

– Batch OS

User programs are collected together (“off line”) and submitted to the OS in a batch by an operator

Print-out results of the program execution is returned to the user

This type of OS is not typical of current machines

This type is used in the mainframes of 60-70s when the cost of hardware was such that you wanted to keep it busy all thetime

O/S was really a monitor program that focused on job scheduling

- Multiprogramming / time sharing

- An OS is said to be multiprogramming if it supports the simultaneous execution of more than 1 job. In this case a queue of pending jobs is maintained

- Current job is swapped out when it is idled waiting for I/O and other devices

Next pending job is started

- Time sharing is the term formultiprogramming when applied to an interactive system

Either requires much more sophistication than a typical batch OS: Memory management and Scheduling

2. OS scheduling

- In multiprogramming OS, multiple jobs are held in memory and alternate between using the CPU, using I/O, and waiting (idle)

- The key to high efficiency with multiprogramming is effective scheduling

– High-level

– Short-term

– I/O

- High-level scheduling

– Determines which jobs are admitted into the system for processing

– Controls the degree of multiprocessing

– Admitted jobs are added to the queue of pending jobs that is managed by the short-termscheduler

– Works in batch or interactive modes

- Short-term scheduling

– This OS segment runs frequently and determines which pending job will receive the

CPU's attention next

– Based on the normal changes of state that a job/process goes through

– A process is running in the CPU until:

+It issues a service call to the OS (e.g., for I/O service)

+ Process is suspended until the request is satisfied. Process causes and interrupt and is Suspended. External event causes interrupt

– Short-term scheduler is invoked to determine which process is serviced next.

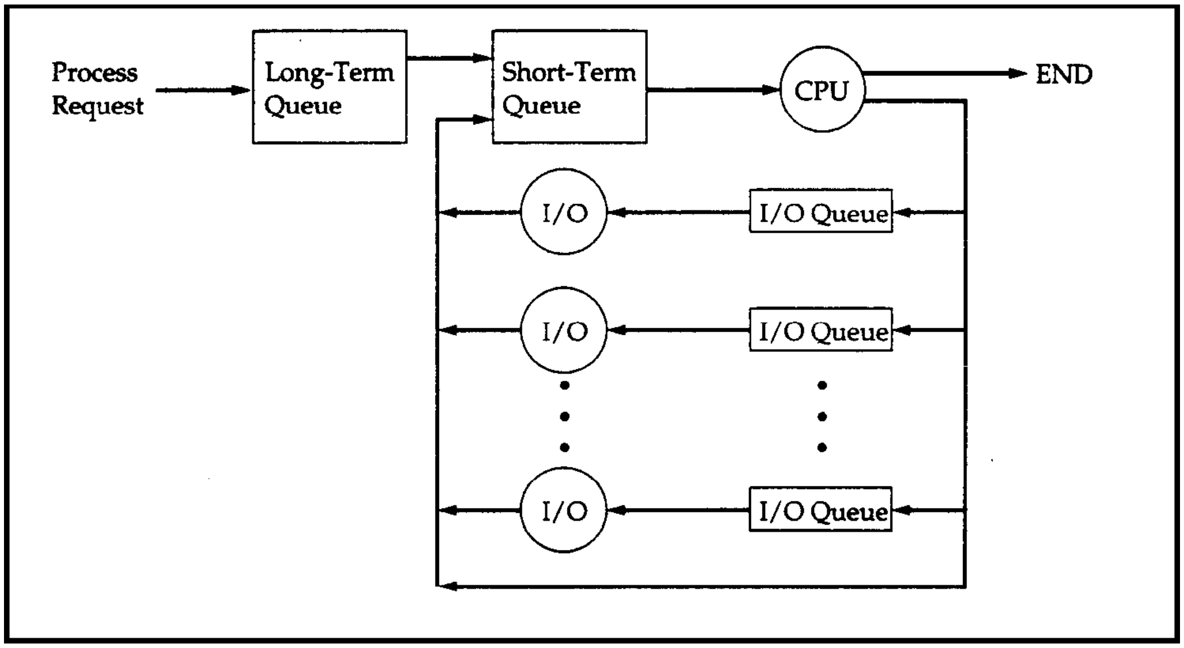


Figure 14.1 Scheduling queues

- Queues are simply lists of jobs that are waiting for some type of service (e.g., CPU cycles, I/ etc.). Once job has been admitted into the system by the high-level scheduler, it is allocated its portion of memory for its program and data requirements

- The objective of the short-term scheduler is to keep the CPU actively working on one of the pending jobs -- maximize the used CPU cycles

- Problem:

+ The high-level scheduler will admit as many jobs as possible (constrained by

system resources such as available memory)

+ Despite many jobs in the system, the speed of the CPU (compared to peripherals) might be such that all jobs are waiting for (slow) I/O and thus the CPU is idled (no jobs in the ready queue)

+ Use memory management to solve this

3. OS Memory Management

One of important task of OS is to menage the memory system. To avoid having the CPU idle because all jobs are waiting, one could increase the memory size to hold more jobs

– Memory is expensive costly solution to the problem

– Programs' sizes (process sizes) tend to expand to take up available memory

A better approach is to use a mechanism to remove a waiting job from memory and

replace it with one that (hopefully) will run on the CPU in the short term

– This is the idea of swapping

– Idle processes are removed from the memory and placed in an intermediate queue

– Replace idled job with a "ready" one from the front of the intermediate queue or with a new job

– Swapping uses I/O operations to read and write to disk.

The virtual memory will be presented in the next section.

Module 15: Virtual Memory

1. Partitioning

Partitioning of memory is as a job that is brought into memory, it must be allocated a portion of the memory, this is its partition. Partitions can be fixed size of variable sized

– Fixed size:

This is a rigid size divisions of memory

Process is assigned to smallest available partition it fits into

Process can be very wasteful of memory

– Variable size

Allocate a process to only as much memory as it needs

Other memory can be used for other processes

Efficient use of memory but, over time, can lead to large numbers of little left over pieces of free memory -- too small to use

Must consider use of memory compaction

- Effect of partitioning

Since a process can be swapped in and out of memory and end up in different partitions, addressing within the process must not be tied to specific physical memory addresses

– Program/process addresses are logical ones with respect to the starting address of the

program itself

– Processor hardware must convert the logical address reference of the program into a physical address that the machine can operate on

– In effect this is a form of index addressing -- base addressing -- where the base address is the first address of the program.

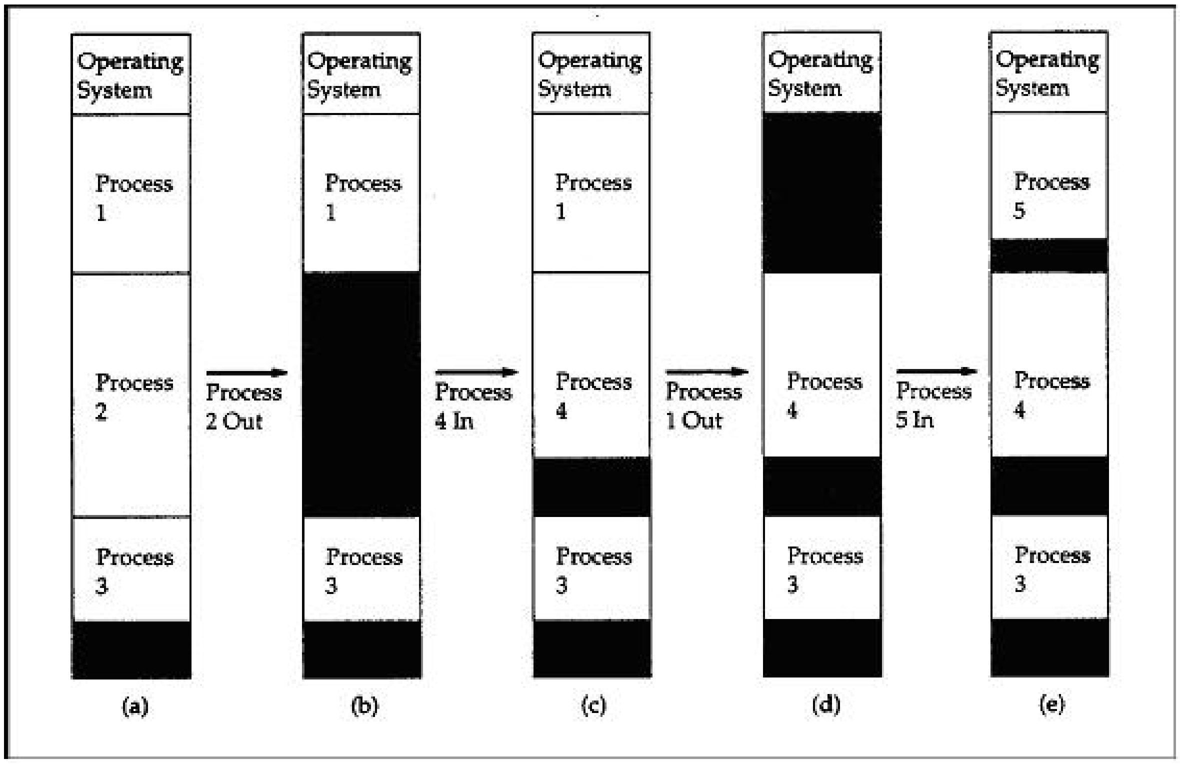


Figure 15.1 Effect of partitioning

2. Paging

- Paging is to subdivide memory into small fixed-size “chunks” called frames or page frames
- Divide programs into same sized chunks, called pages
- Loading a program in memory requires the allocation of the required number of pages
- Limits wasted memory to a fraction of the last page
- Page frames used in loading process need not be contiguous

– Each program has a page table associated with it that maps each program page to a memory page frame

– Thus, logical addresses in a program are interpreted as a physical address -- page frame number and an offset within the page

- Principle for allocation of free page frames

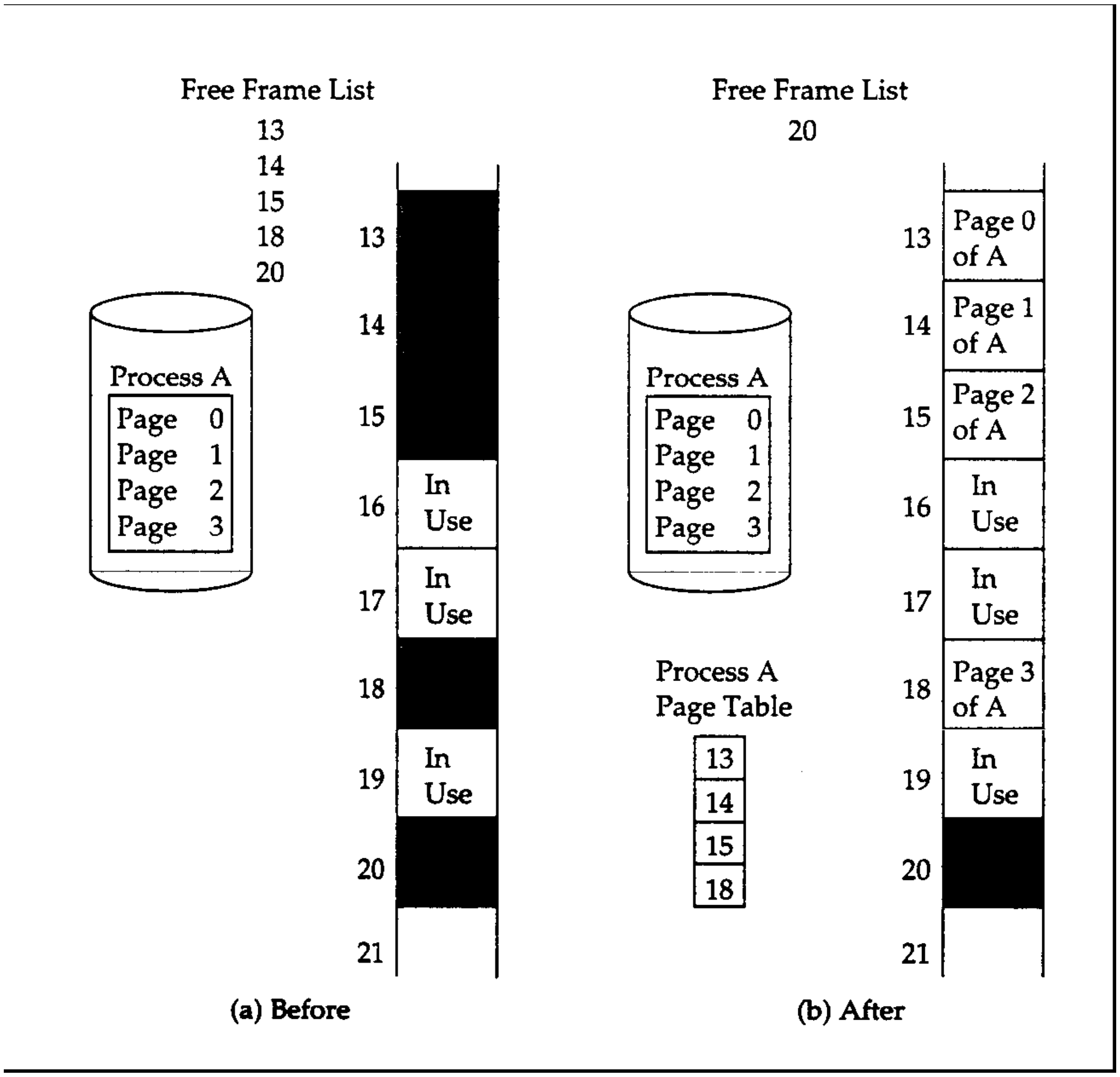


Figure 15.2 Allocation of free pages

- Address mapping in a paging system

The more sophistiated way of mapping address from the address space onto the actual memory address is possible. Principle of address mapping in a paging system is shown in Figure 15.3 .

The address that the program can refer is called the virtual address space, the actual hardwired (pgysical) memory address is called the physical address space.

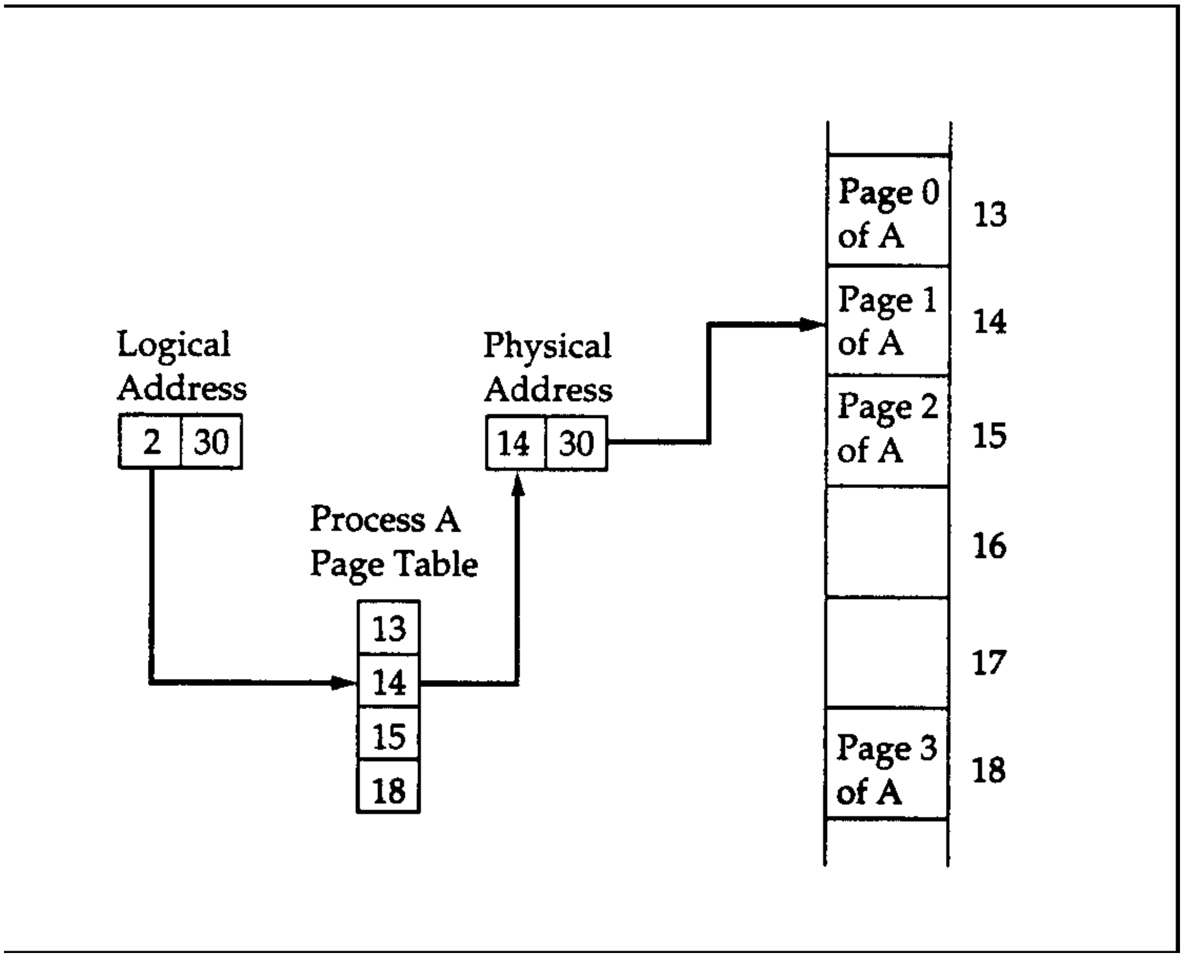


Figure 15.3 Address mapping in a paging system

3. Implementation of Paging

- The implementation of paging systems permit true multiprogramming systems through the use of virtual memory / demand paging

– Only the program pages that are required for the execution of the program are actually loaded -- hence the “demanded” concept

– Only a few (of the many potential) pages of any one program might be in memory at a time

– Many “active” programs can have pages -- increasing the degree of multiprogramming

- With demand paging, it is possible for a program consisting of more pages than can (ever) fit into memory to run

– This is presents a virtual memory system that seems unlimited in size to the programmer

– Significantly simplifies program development

- Page tables can be large and are themselves subject to being stored in

secondary storage. Only a small portion will be kept in main memory

- To reduce size of a page table consider

– Using a 2-level page table:

Entry in first level point to a table at the second level

First level is small -- in many cases the size of a single page

– Use an inverted table structure: A hashing function maps a page request

into an inverted table that is used by all processes. One structure is usec for all processes and virtual pages

- It would seem that each virtual address reference causes 2 memory accesses

– An access to get to the page table

– An access to get the actual data page

- To avoid this delay, use a special cache to hold page table information
- Every memory access requires

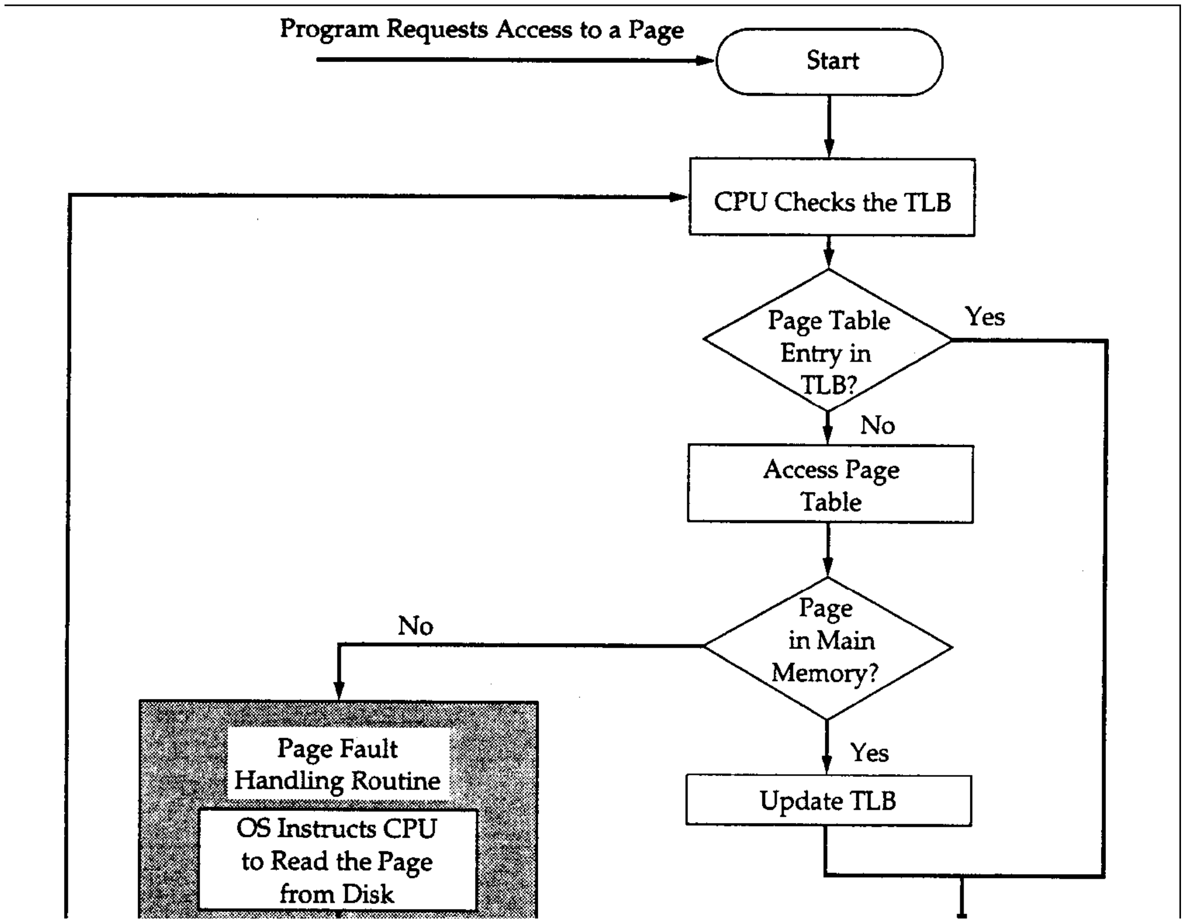
– Trying to find frame reference in TLB

– If not there, go to page table in memory

– With frame identified, check the cache to see if it is there,

– If not there, get from main memory or retrieve from disk

- Operation of paging and TLB



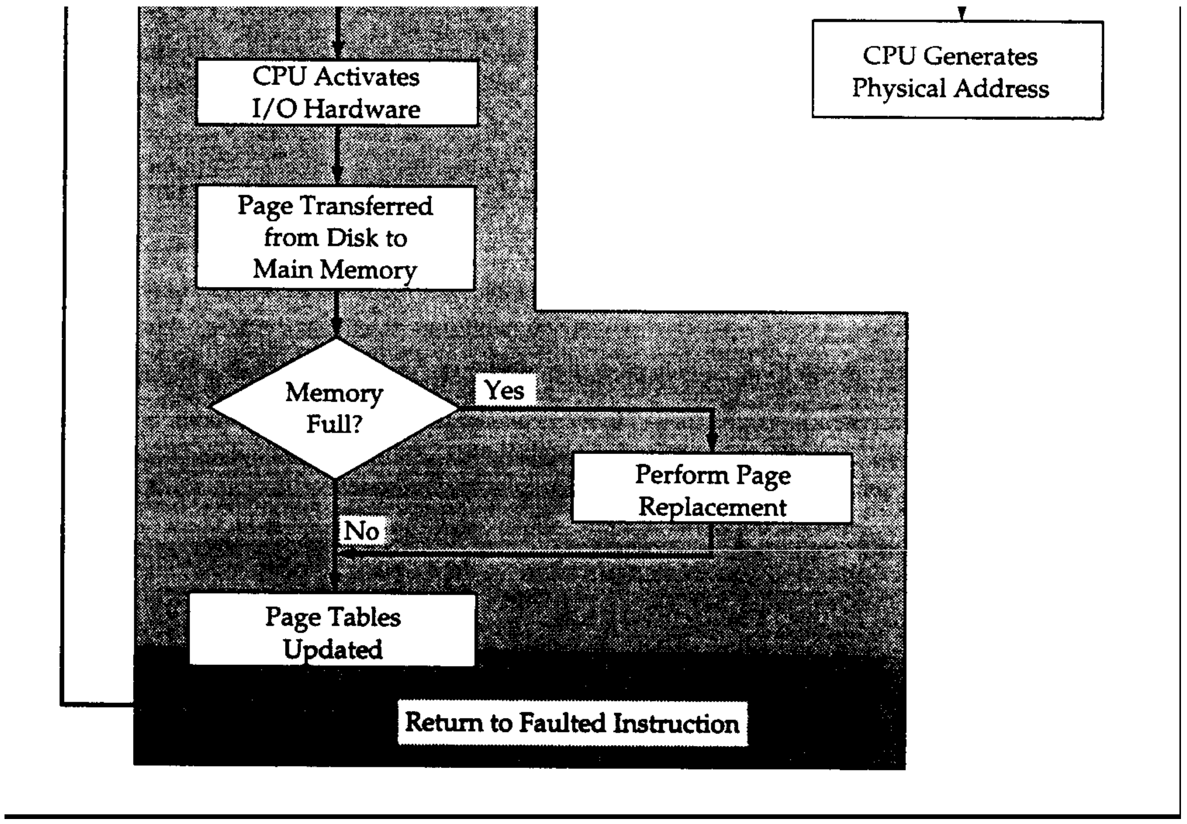

CPU Activates I/O Hardware
Page Transferred from Disk to Main Memory
Memory Full?
Yes
Perform Page Replacement
No
Page Tables Updated
Return to Faulted Instruction
CPU Generates Physical Address

Figure 15.3 Operation of paging and TLB

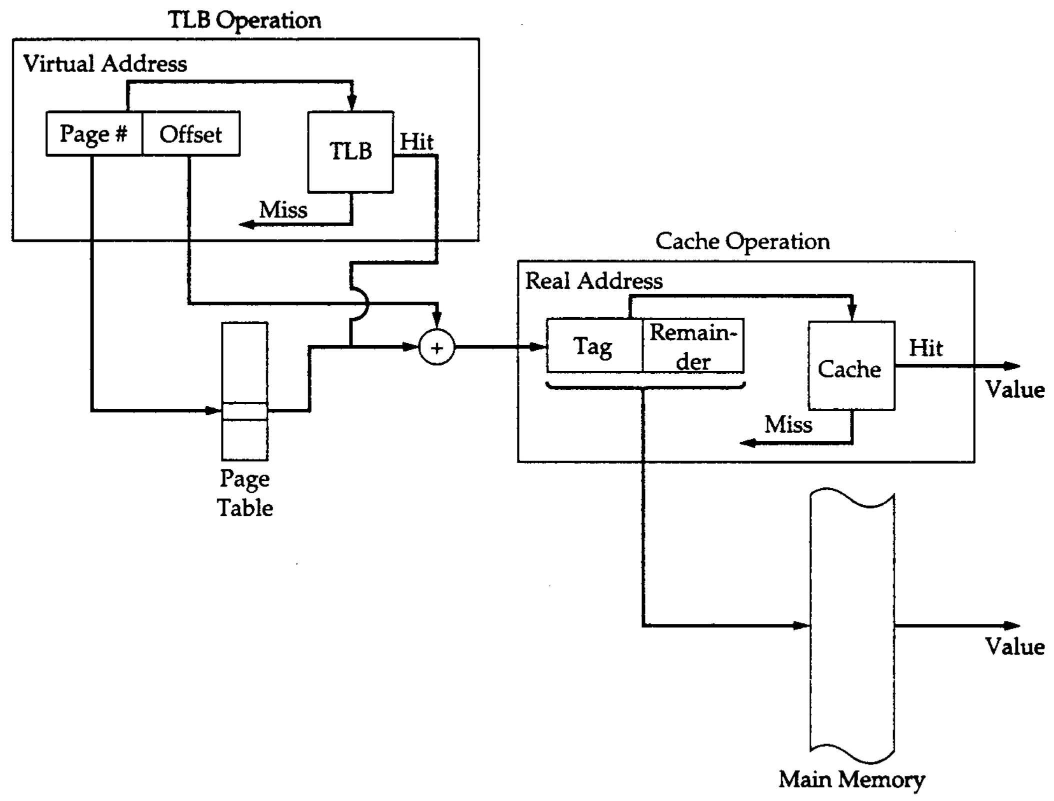


Figure 15.4. TLB and cache operation

4. Page Replacemenr Policy

- Principle of Replacemenr Policy

- If there is an unused page frame, it can be loaded into it

- If all page frames are in use, one must be selected for replacement

- If it has been altered, the page to be replaced must be copied back to secondary

storage

- Page replacement algorithms

Similar to the ones discussed in caching, the following algorithms have been used

LRU

FIFO

Random, etc.

5. Segmentation

In addition to partitioning and paging, segmentation can be used to subdivide

memory

– Programmer subdivides the program into logical units called segments

Programs subdivided by function

Data array items grouped together as a unit

– Advantages

Simplifies data structure handling

Supports code modification and maintenance

Supports sharing code segments

Provides a degree of protection

– Memory reference is in the form of a segment ID and an offset

– Segmentation can be combined with paging in a segment ID, page number, page offset scheme

Resume of Virtual Memory management

– Partitioning

– Paging

– Segmentation

– Segmentation and paging

Module 16: Advanced Architectures

1. Introduction

In the previous sections of this course. we have concentrated on singleprocessor architectures and techniques to improve upon their performance, such as:

– Efficient algebraic hardware implementations

– Enhanced processor operation through pipelined instruction execution and multiplicity of functional units

– Memory hierarchy

– Control unit design

– I/O operations

Through these techniques and implementation improvements, the processing power of a computer system has increased by an order of magnitude every 5 years. We are (still) approaching performance bounds due to physical limitations of the hardware.

- Several approaches of parallel computer are possible

– Improve the basic performance of a single processor machine

Architecture / organization improvements

Implementation improvements

SSI --> VLSI --> ULSI

Clock speed

Packaging

– Multiple processor system architectures

Tightly coupled system

Loosely coupled system

Distributed computing system

- Parallel computer: SIMD computer, MIMD computer

2. Multiple Processor Systems

System with multiprocessor CPUs can be divided into multiprocessor and multicomputers. In this section we will first study multiprocessors and then multicomputers

Shared-Memory Multiprocessor

A parallel computer in which all the CPUs share a common memory is called a tightly coupled systems

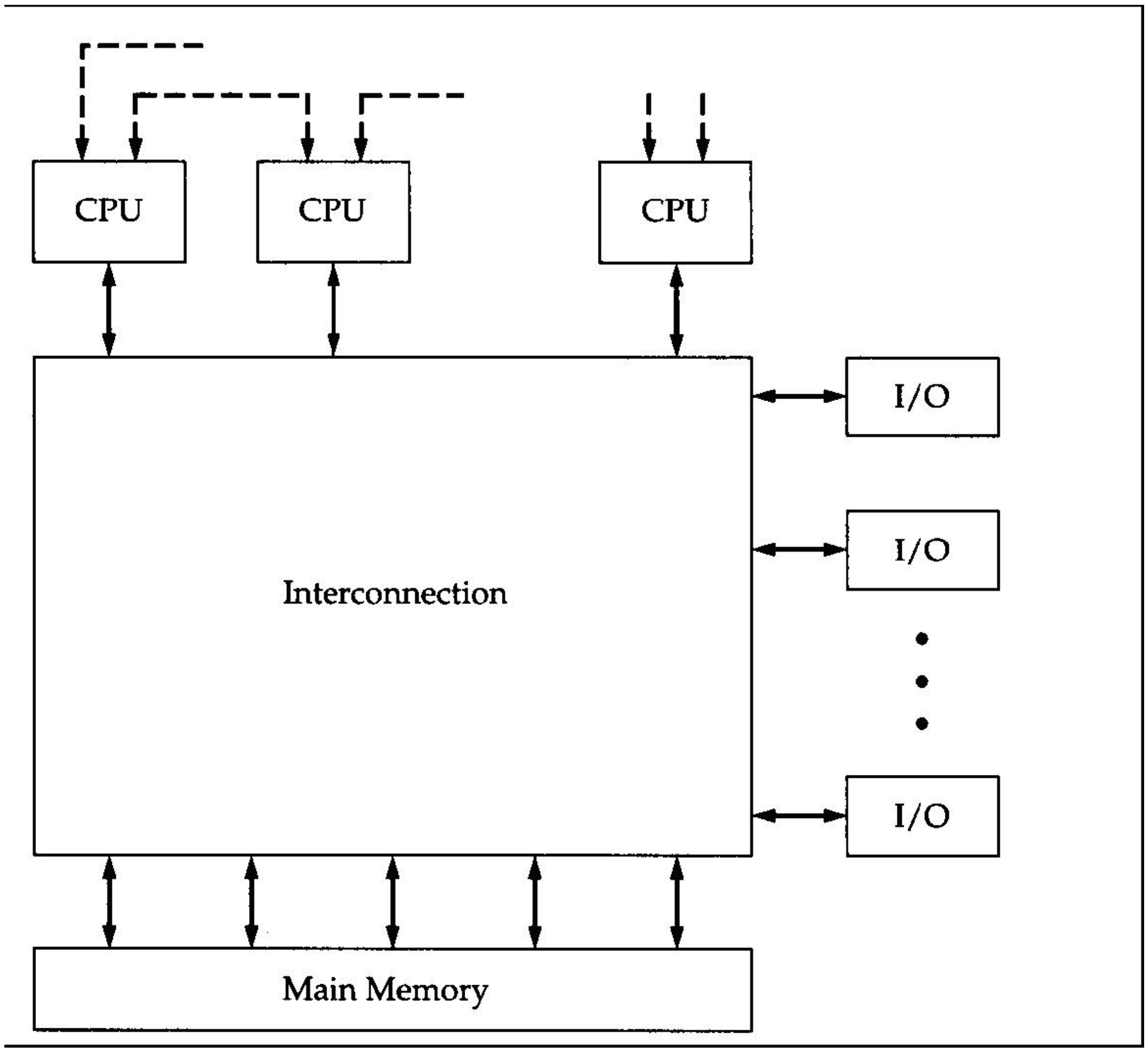


Figure 16.1. Tightly coupled systems, Shased-memory multiprocessor

- The features of the system are as follow.

– Multiple processors

– Shared, common memory system

– Processors under the integrated control of a common operating system

– Data is exchanged between processors by accessing common shared variable locations in memory

– Common shared memory ultimates presents an overall system bottleneck that effectively limits the sizes of these systems to a fairly small number of processors (dozens)

Message-passing multiprocessor

A parallel computer in which all the CPUs has a local independent memory is called a loosely coupled systems

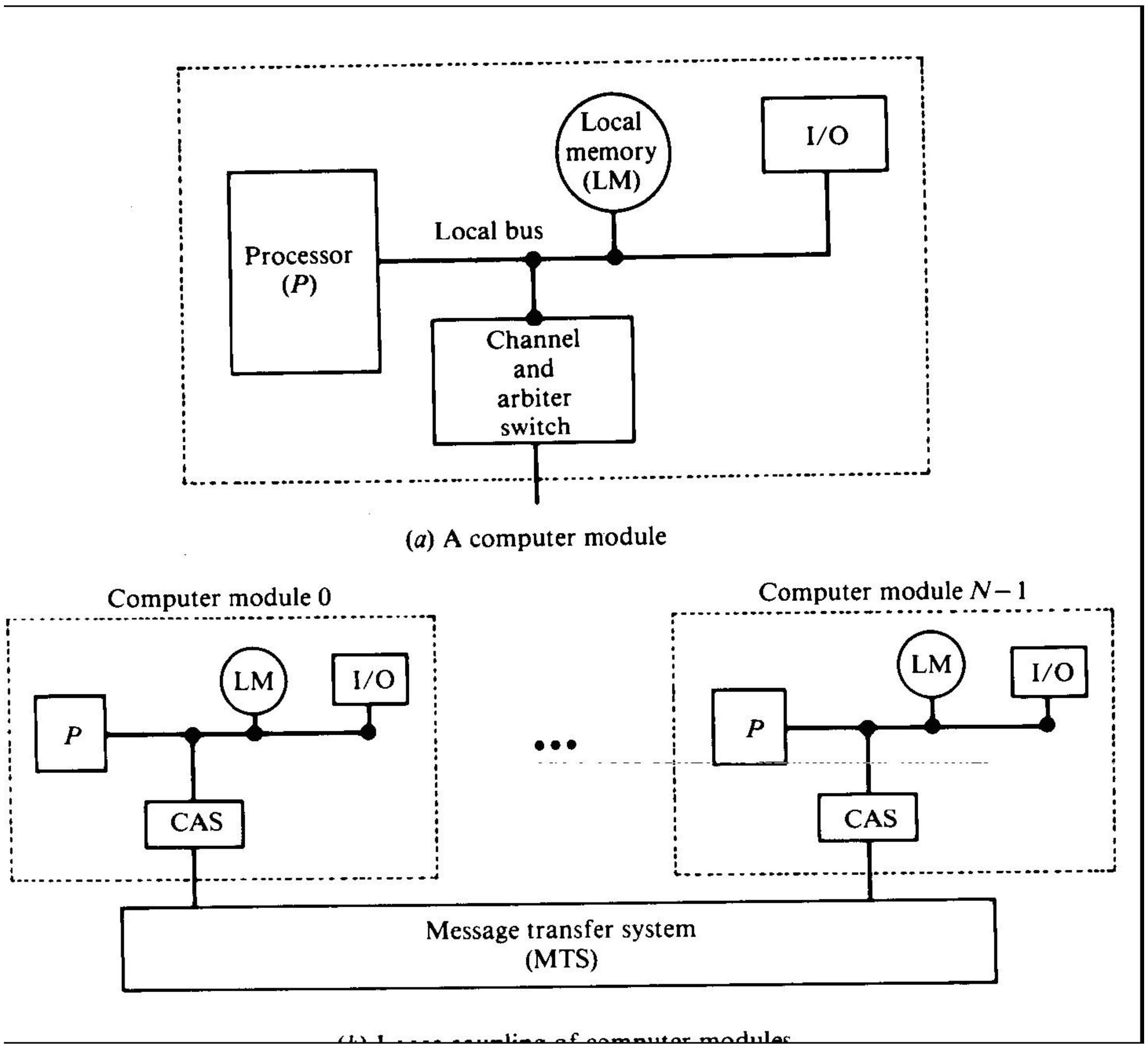


Figure 16.2. Loosely coupled systems, Message-passing multiprocessor

- The features of the system are as follow.

– Multiple processors

– Each processor has its own independent memory system

– Processors under the integrated control of a common operating system

– Data exchanged between processors via interprocessor messages

– This definition does not agree with the one given in the text

Distributed computing systems

Now we can see the message-passing computer that multicomputer are held togerther by network.

– Collections of relatively autonomous computers, each capable of independent operation

– Example systems are local area networks of computer workstations

+ Each machine is running its own "copy" of the operating system

+ Some tasks are done on different machines (e.g., mail handler is on one machine)

+ Supports multiple independent users

+ Load balancing between machines can cause a user's job on one machine to be shifted to another

Performance bounds of Multiple Processor Systems

- For a system with n processors, we would like a net processing speedup (meaning lower overall execution time) of nearly n times

when compared to the performance of a similar uniprocessor system
- A number of poor performance “upper bounds” have been proposed over the years

Maximum speedup of O(log n)

Maximum speedup of O(n / ln n)

- These “bounds” were based on runtime performance of applications and were not necessarily valid in all cases
- They reinforced the computer industry’s hesitancy to “get into” parallel processing

3. Parallel Processors

The machines are the true parallel processors (also called concurrent processors)

These paralle machines fall into Flynn’s taxonomy classes of SIMD and MIMD systems

– SIMD: Single Instruction stream and Multiple Data streams

– MIMD: Multiple Instruction streams and Multiple Data streams

SIMD Overview

- Single “control unit” computer and anarray of “computational” computers
- Control unit executes control-flowinstructions and scalar operations and passes vector instructions to the processor array
- Processor instruction types:

– Extensions of scalar instructions

Adds, stores, multiplies, etc. become vector operations executed in all processors concurrently

– Must add the ability to transfer vector and scalar data between processors to the instruction set -- attributes of a “parallel language”

- SIMD Examples

Vector addition

C(I) = A(I) + B(I)

Complexity O(n) in SISD systems for I=1 to n do

C(I) = A(I) + B(I)

Complexity O(1) in SIMD systems

Matrix multiply

A, B, and C are n-by-n matrices

Compute C= AxB

Complexity O(n3) in SISD systems

n2 dot products, each of which is O(n)

Complexity O(n2) in SIMD systems

Perform n dot products in parallel across M the array

Image smoothing

– Smooth an n-by-n pixel image to reduce “noise”

– Each pixel is replaced by the average of itself

and its 8 nearest neighbors

– Complexity O(n2) in SISD systems

– Complexity O(n) in SIMD systems

Pixel and 8 neighbors

MIMD Systems Overview

- MIMD systems differ from SIMD ones in that the “lock-step” operation requirement is removed
- Each processor has its own control unit and can execute an independent stream of

instructions

– Rather than forcing all processors to perform the same task at the same time, processors can be assigned different tasks that, when taken as a whole, complete the assigned application

- SIMD applications can be executed on an MIMD structure

– Each processor executes its own copy of the SIMD algorithm

- Application code can be decomposed into communicating processes

– Distributed simulations is a good example of a

very hard MIMD application

- Keys to high MIMD performance are

– Process synchronization

– Process scheduling

- Process synchronization targets keeping all processors busy and not suspended awaiting data from another processor

- Process scheduling can be performed

– By the programmer through the use of parallel language constructs

Specify apriori what processes will be instantiated and where they will be

executed

– During program execution by spawning processes off for execution on available processors.

Fork-join construct in some languages

- System examples

SIMD

– Illiac IV

One of the first massively parallel systems 64 processors

– Goodyear Staran: 256 bit-serial processors

– Current system from Cray Computer Corp.uses supercomputer (Cray 3) front end coupled to an SIMD array of 1000s of processors

MIMD

– Intel hypercube series:

Supported up to several hundred CISC processors

Next-gen Paragon

– Cray Research T3D

Cray Y-MP coupled to a massive array of Dec Alphas

Target: sustained teraflop performance

4. Discussions

- Problems

– Hardware is relatively easy to build

– Massively parallel systems just take massive amounts of money to build

– How should/can the large numbers of processors be interconnected

– The real trick is building software that will exploit the capabilities of the system

- Reality check:

– Outside of a limited number of high-profile applications, parallel processing is still a“young” discipline

– Parallel software is still fairly sparse

– Risky for companies to adopt parallel strategies, just wait for the next new SISD system.